JTC1/SC22/WG21 N1850=05-0110
2005-08-25

Towards a Better Allocator Model

Pablo Halpern <phalpern@halpernwightsoftware.com>

ABSTRACT

In drafting the 1998 ISO C++ language standard, the standards committee did not consider, or did not consider important, that the allocator type parameter used to instantiate a container template affects the container's type, thereby making it incompatible with containers employing a different allocation policy, but which otherwise have identical type. In our work, we have found it desirable, and often necessary, to customize the memory allocation for our containers and strings on a per-instance basis, without affecting the compile-time type of those containers or strings. Proper control over memory also required that objects within a container share an allocator with the container itself. Using an existing implementation of the C++ standard library as a base, we made backward-compatible modifications to enable such per-instance allocators. Our experience is that, once implemented, these modifications allowed us to use allocators in a wide range of situations to facilitate testing, efficient memory utilization, and even the prevention of memory leaks. This paper describes our allocator model and proposes changes to the upcoming revision of the standard library based on our experience.

Copyright and Disclaimer

Table of Contents

1 Introduction

1.1 History

Before the STL portion of the C++ Standard Library was adopted by our organization, we began creating our own assortment of containers. Like the STL containers, our containers could be fitted with allocators to customize their use of memory. At the most conceptual level, an ***allocator*** is an object that supplies raw memory for use by other objects, especially containers. The specifics of our container/design scheme, however, were different from STL's design in a number of important ways, as described in the succeeding sections of this paper.

As we moved to more modern compilers that were capable of handling the demands of STL templates, we decided to migrate away from our proprietary containers and towards STL in order to improve interoperability with other software libraries and avoid maintaining a redundant set of classes. In the process of migrating, we discovered that we were unable to cleanly adapt our use of allocators to the standard allocator scheme. In particular, we needed to be able to customize memory allocation for a specific instance of a container or string without affecting the type of that container or string (through an allocator template parameter). Our allocator semantics were simply more powerful and flexible than the standard's and were based on theoretical principles that we believed to be more cohesive.

We seriously considered abandoning STL containers in order to retain the benefits of our allocator design, but the benefits of interoperability and standard-compliance were too great to give up. In the end, we decided to extend an existing implementation of the C++ Standard Library so that it would support both the standard allocator semantics and our allocator semantics. Having successfully accomplished this marriage and reaping the benefits, we believe that our extensions deserve serious consideration for adoption into the next C++ standard.

1.2 Terminology

Templates introduce a meta level into the C++ language. That is, they don't describe run-time constructs but rather describe classes and functions, which, in turn, describe run-time constructs. As with all discussions of meta concepts, the terminology can get confusing. In this paper, I am concerned mostly with containers and allocators. Yet the term, ***container*** is ambiguous; it might refer to a class template, a specific instantiation of a class template, or an object created from such an instantiation. I will attempt to avoid confusion by using the terms ***container template***, ***container instantiation***, or ***container object*** whenever an ambiguity may arise. Similarly, I use the terms ***allocator template***, ***allocator instantiation***, or ***allocator object*** whenever ***allocator*** alone might be ambiguous. For example, in the following definition:

```
std::vector<int> v;
```

I will refer to the parts as follows:

`std::vector`	is a *container template,*
`std::vector<int>`	is a *container instantiation,* and
`v`	is a *container object.*

The standard introduces a second level of meta-concept in the form of ***requirements***. Requirements specify the interface for a family of templates, which can be used to create an unbounded set of instantiations, which in turn can be used to construct an unbounded set of objects.

I use the term ***STL*** to refer to the containers and allocators section of the C++ Standard Library – those portions that have their origins in the Standard Template Library created by Alexander Stepanov et al. [Stepanov95]. Although character strings were not originally part of the STL, I include them when I talk about STL because they share the qualities of STL containers.

2 The Lakos Allocator Model

The allocators and containers described in this section form part of a project known as BDE (Basic Development Environment) at Bloomberg LP. The allocator model presented here was brought to my attention by fellow Bloomberg employee John Lakos. Most of the work of unifying his allocator model with the STL allocator model was performed by me at Bloomberg, and will be described later in this paper.

2.1 The bde::Allocator Class Hierarchy

If you are already very familiar with the STL approach to allocators, I ask you to set that knowledge aside for a moment as I discuss the way BDE allocators work. An allocator type in BDE is a class derived from the `bde::Allocator` abstract base class, which has the following interface:

```
class bde::Allocator {
  public:
    virtual void* allocate(size_t bytes) = 0;
    virtual void deallocate(void* ptr) = 0;
    virtual ~Allocator();
};
```

This allocator interface is similar to those defined by a number of different companies in the pre-STL days and even today (for example, in the Xerces XML parser).

BDE supplies a number of allocator classes for various purposes:

New/delete allocator	Allocates memory using `operator new` and frees it using `operator delete`.
Shared-memory allocator	Allocates memory from a shared memory region.
Limit allocator	Keeps track of the number of bytes that have been allocated from the allocator. Throws an exception if too much memory is used. Useful for preventing denial-of-service attacks that would otherwise cause excessive memory consumption.
Buffer Allocator	Allocates memory from a fixed-sized buffer provided at construction.
Arena Allocator	Very fast allocator in which `allocate` simply increments a pointer within a large, contiguous block of memory and in which `deallocate` is a no-op. This allocator saves time by avoiding the bookkeeping needed to free individually-allocated blocks of memory. The allocator's destructor releases one or more large blocks of memory to the heap all at once.
Test Allocator	Keeps track of allocated bytes and blocks. Checks for memory leaks. Can be configured to throw an exception on the ***n***th allocation attempt. Useful in unit-test drivers to compare actual memory use against expected memory use and to test exception safety of components that allocate memory.

Some of these allocator types can be chained together so that, for example, a limit allocator can manage memory provided by a shared-memory allocator.

2.2 Per-instance Allocators

Every constructor declared in each container class template takes an optional allocator pointer argument. For example, the `bde::Array` template, which is roughly equivalent to `std::vector`, has constructors declared like this:

```
template <typename T>
class bde::Array {
  public:
    Array(bde::Allocator* allocator = 0);
    Array(const Array& rhs, bde::Allocator* allocator = 0);
    // ...
};
```

The container obtains the memory it needs for its internal data structures by calling the allocator's `allocate` function and releases it when it is done by calling the allocator's `deallocate` function. The default arguments allow the first constructor to be recognized as the default constructor and the second constructor to be recognized as the copy constructor. If no allocator pointer is passed to the constructor, the container uses the default allocator, which is the instance of the new/delete allocator returned by a static singleton function.

With the exception of the new/delete allocator, all of the allocators listed above contain state information that differs from instance to instance. This means, for example, that two buffer allocator instances will manage memory from two different buffers. The BDE allocator mechanism was designed to support ***per-instance allocators***, whereby two container objects of the same type can get their memory from different sources.

The allocator pointer, which is held (but not owned) by each container (or string), is an implementation mechanism and is not part of the container's value. Concretely, the allocator pointers' not being part of the value means that they are not tested as part of the container's operator==, nor copied in its copy constructor or assignment operator. This clean separation between a container's value and its allocation mechanism is key to effective use of allocators in practice.

2.3 Contained-Element Allocators

A container will often contain other containers. One can have, for example, a `vector` of `vectors`, or a `set` of `strings` (a string having all of the qualities of a container of `char`). In general, we want all of the parts of such compound containers to get their memory from a common source. Thus, a critical feature of the Lakos allocator model is the automatic use of a container's allocator to construct its contained elements. When an object is inserted into a container, the address of the container's allocator is passed as a second argument to the object's copy constructor (see `bde::Array` example, previously). A `bde::Array` of `bde::String` (think `vector` of `string`) can be visualized as in Figure 1.

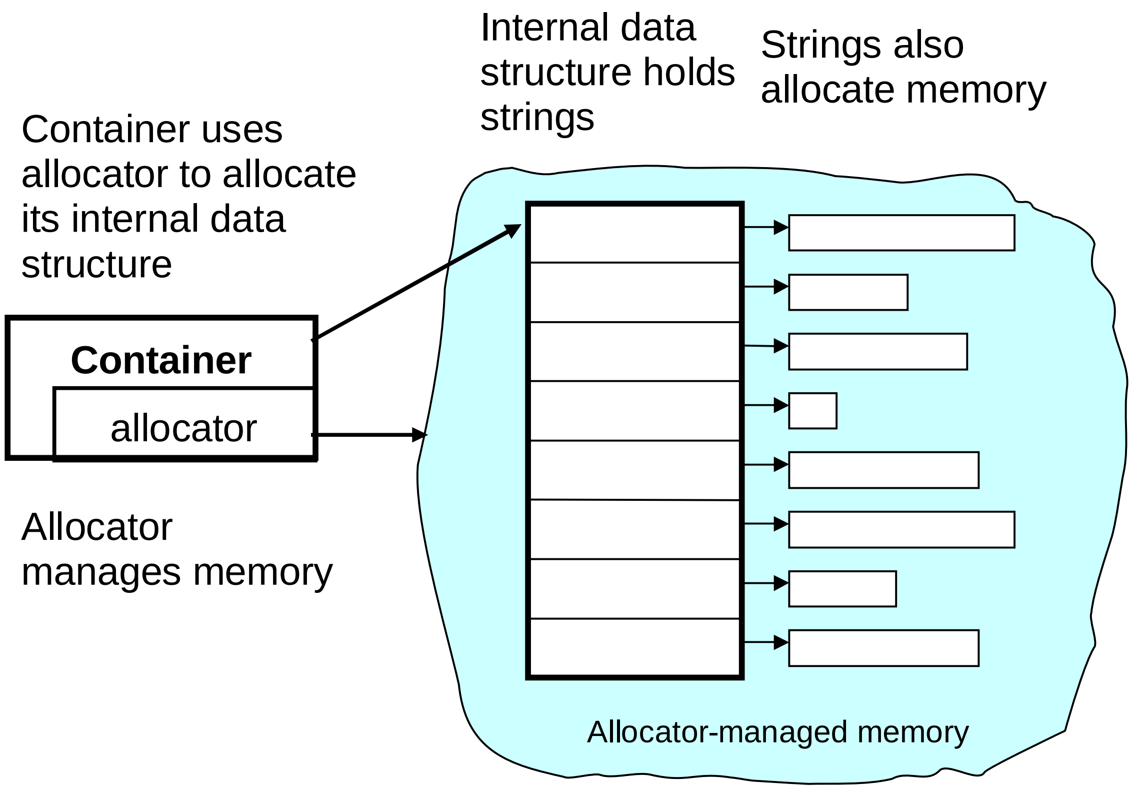
Internal data structure holds strings
Strings also allocate memory
Container uses allocator to allocate its internal data structure
Container
allocator
Allocator manages memory
Allocator-managed memory

Figure 1: A container and its contained objects use the same allocator

The challenge of this approach is that the container template must be able, at compile time, to determine whether it is being instantiated with an element type that uses allocators. For example, a `bde::Array<int>` cannot pass its allocator to the `int` constructor, whereas `bde::Array<bde::String>` must pass its allocator to the `bde::String` constructor. In order to give the container knowledge of its contained element type, we introduced a trait, similar in concept to Boost traits [Maddock01] and TR1 traits [TR1]. The new trait is called `bde::UsesBdeAllocator` and applies to each class that takes an (optional) BDE-style allocator pointer argument for both its default and copy constructors. Container types take advantage of generic algorithms, which work differently depending on whether their argument does or does not have this trait. For example, code to insert a new element into a container looks something like this:

```
template <typename T>
void bde::Array::append(const T& value) {
    //...
    T* newElement = elements + length();
    copyConstruct(newElement, value, getAllocator());
}
```

The `copyConstruct` algorithm is declared as follows:

```
template <typename T>
void copyConstruct(T* p, const T& orig, bde::Allocator* alloc);
```

If type `T` has the `bde::UsesBdeAllocator` trait, then this function's implementation resolves to:

```
new (p) T(orig, alloc);
```

Otherwise, it resolves to:

```
new (p) T(orig);
```

A `defaultConstruct` function template works the same way, except without the `orig` argument. The use of traits classes to selectively instantiate different algorithm implementations is covered in depth in [Alexandrescu01].

2.4 Allocators as an Extension of Storage Class

When the C language was standardized in 1989, the standardization committee was careful to distinguish the concept of ***storage duration*** (static, automatic, and dynamically-allocated) from the concept of ***type qualifier*** (`const` and `volatile`). Although both concepts may be part of an object's definition (storage duration is sometimes explicitly specified via an `extern`, `static`, or `auto` ***storage class*** specifier), a type qualifier alters the type of an object, whereas its storage duration does not. There is no need to think twice before initializing an automatic object with the ***value*** of a static object of the same type. There is no concern that objects might not be compatible because they have different storage duration, nor is there a concern that the local copy of the object becomes static simply because it was initialized from a static value.

A general principle that has emerged from our examination of allocators is that an allocator should be more like a storage class than like a type qualifier. Specifically, the following principles hold for Lakos-model allocators:

1. **An object's allocator (as supplied to its constructor) does not change during that object's lifetime**.

2. **The allocator supplied to an object does not affect that object's type.**

3. **Allocators are not transferred implicitly on copy-construction.** Instead, an allocator can be supplied explicitly to the copy. The copy uses the default (new/delete) allocator if one is not supplied.

4. **Containers supply their own allocator to the objects they contain.** This principle provides a clean semantic and is analogous to the C precept that array elements have the same storage class as the array itself.

The overall allocator system used in BDE is consistent. The allocator is neither part of the type nor part of the value of a container or string. The resulting system has proven itself in practice to be extremely easy to use and to extend.

3 The 1998 Standard Allocator Model

3.1 Allocator Class Templates

The allocator model described in the C++ Standard is very different from the one I have just described. All of the container templates described in the standard library have an optional template parameter for specifying an optional allocator type. For example, the `list` class template is declared like this:

```
template <typename T, typename Alloc = std::allocator<T> > class list;
```

The standard sets forth ***allocator requirements*** and ***container requirements*** that describe minimal interfaces for allocator templates and container templates. The standard also provides a specific allocator template (`std::allocator<T>`), and a number of container templates (`std::vector<T,Alloc>`, `std::set<T,Comp,Alloc>`, etc.) A programmer may supply his/her own allocator template for use with the standard containers, provided it adheres to the allocator requirements. Similarly, a programmer may create additional container templates that adhere to the container requirements.

If we wish to control the way in which `list` uses memory, we can create our own allocator template that meets the requirements set in the Standard:

```
template <typename TYPE> class MyAllocator
{
  public:
    typedef TYPE           value_type;
    typedef TYPE*          pointer;
    typedef const TYPE*    const_pointer;
    typedef TYPE&          reference;
    typedef const TYPE&    const_reference;
    typedef std::size_t    size_type;
    typedef std::ptrdiff_t difference_type;

    template <typename U> class rebind
    {
        typedef MyAllocator<U> other;
    };

    MyAllocator();
    template <typename U>
        MyAllocator(const MyAllocator<U>& y);

    ~MyAllocator();

    pointer address(reference x) const { return &x; }
```

```
        const_pointer address(const_reference x) const { return &x; }

        TYPE* allocate(size_type n, const void* hint = 0);
        void deallocate(TYPE* p, size_type n);
        size_type max_size() const;

        void construct(pointer     p,
                       const TYPE& t) const { new ((void*) p) TYPE(t); }
        void destroy(pointer p) const { p->~TYPE(); }
    };

    template <typename T1, typename T2>
    bool operator==(const MyAllocator<T1>& a1, const MyAllocator<T2>& a2);
    template <typename T1, typename T2>
    bool operator!=(const MyAllocator<T1>& a1, const MyAllocator<T2>& a2);
```

Although `MyAllocator` is not part of the standard, anybody intending to write a standard-compliant allocator would do well to copy it, change its name, and implement those member functions whose implementation is not already provided. The core of each allocator template – the thing that makes it different from other allocator templates – is the `allocate` and `deallocate` functions. The `allocate` and `deallocate` functions manage ***raw*** memory and do not call `TYPE`'s constructor or destructor. Even so, the *n* argument to `both functions` is a count of ***objects*** being managed, not the number of ***bytes***. Instantiating `list` as `std::list<int, MyAllocator<int> >` will cause `list` to use the custom allocator mechanism instead of the default one.

3.2 Rebind

Let me draw your attention to the `rebind` class and to the templated constructor in `MyAllocator`. Most containers, with the possible exception of `vector` and `deque`, do not directly use the allocator with which they are instantiated. The `list` template, for example, will typically be implemented as a linked list of nodes, where each node is an instantiation of a template class that looks something like this:

```
    template <typename T> struct ListNode {
        ListNode* prev;
        ListNode* next;
        T         data;
    };
```

A `list` of `int`s would not need to allocate objects of type `int` but rather `ListNode<int>`. Unfortunately, within our `list` instantiation, the name `Alloc` is bound to `MyAllocator<int>` and cannot be used directly to allocate `ListNode<int>` objects. That is where `rebind` comes in. The type, `typename Alloc::template rebind<ListNode<T> >::other`, will resolve to `MyAllocator<ListNode<T> >`, which is exactly what is needed within `list`. The templated constructor of `MyAllocator` lets us construct a `MyAllocator<ListNode<T> >` object from a `MyAllocator<int>` object. This clever trick has some interesting consequences, as we will see later.

4 Difficulties with the 1998 Standard Allocator Model

4.1 The Incomplete Standard

According to the standard, two allocator objects of the same type may compare equal only if memory allocated using one can be de-allocated using the other. Section 20.1.5, paragraph 4 of the 1998 standard states that an implementation may assume that all instances of a given allocator type compare equal and are therefore interchangeable. (I refer to this as the ***equal-allocator assumption.***) The constant-time, semantics of `swap` and `list::splice` pose a particular problem when allocators have state and can compare unequal [Issue431]. A conforming program must assume that any memory allocated with one allocator might be de-allocated using a different allocator of the same type, making per-instance allocators effectively useless except perhaps as performance hints. By including a normative note encouraging implementations to specify reasonable behavior in the presence of unequal (per-instance) allocators, the members of the standards committee, to their credit, effectively admitted that the allocator model was incomplete. Also to their credit, they chose not to delay the standard while waiting for a more complete proposal.

At the time that the equal-allocator assumption was added to the 1998 standard, it was almost certain that it would be removed in the next revision of the standard. In fact, even the 1998 standard describes attributes of allocators (such as the meaning of `operator==` and the behavior of copying a container) that have meaning only in the absence of the equal-allocator assumption. For the remainder of this paper, I will take for granted that the equal-allocator assumption will be removed, resulting in an allocator model does allow per-instance allocators, but with significant deficiencies, as described in the next few sections.

4.2 Memory Models

Memory allocators were originally added into the Standard Template Library for the purpose of supporting Intel x86 mixed-model (near and far) pointers and other non-flat memory models [LoRusso01]. The theory was that memory could be allocated from either the local data segment, producing a "near pointer", or from a different memory segment, producing a "far pointer". The `pointer` type declared within the allocator would allow the container to use the results appropriately. In practice, however, `std::allocator<T>::pointer` cannot be other than `T*`, and this realization was codified in the standard itself (section 20.1.5, table 32). Most compilers targeted for the Intel architecture have moved on to the flat memory model anyway.

4.3 Template Implementation Policy

The first problem most people see with the allocator mechanism as specified in the Standard is that the choice of allocator affects the type of a container. Consider, for example, the following type and object definitions:

```
typedef std::list<int, std::allocator<int> > NormIntList;
typedef std::list<int, MyAllocator<int> >    MyIntList;

NormIntList list1(5, 3);
MyIntList   list2(5, 3);
```

`list1` and `list2` are both lists of integers, and both contain five copies of the number 3. Most people would say that they have the same ***value***. Yet they belong to different types and you cannot substitute one for the other. For example, assume we have a function that builds up a list:

```
int build(std::list<int>& theList);
```

Because we did not specify an allocator parameter for the argument type, the default, `std::allocator<int>` is used. Thus, `theList` is a reference to the same type as `list1`. We can use `build` to put values into `list1`, but we cannot use it to put values into `list2` because `MyIntList` is not compatible with `std::list<int>`. The following operations are also not supported:

```
list1 == list2
list1 = list2
MyIntList list3(list1);
NormIntList* p = &list2;
// etc.
```

Now, some would argue that the solution to the `build` function problem is to templatize `build`:

```
template <typename Alloc>
int build(std::list<int, Alloc>& theList);
```

or, better yet:

```
template <typename OutputIterator>
int build(OutputIterator theIter);
```

Both of these templatized solutions have their place, but both add substantial complexity to the development process. Templates, if overused, lead to long compile times and, sometimes, bloated code. If `build` were a template and passed its arguments on to other functions, those functions would also need to be templates. This chained instantiation of templates produces a deep compile-time dependency such that a change to any of those modules would result in a recompilation of a significant part of the system. For thorough coverage of the benefits of reducing physical dependencies, see [Lakos96].

Even if the templatization solution were acceptable, once a nested container (e.g. a list of strings) is involved, even the simplest operations require many layers of code to bridge the type-interoperablity gap. Consider trying to compare a shared list of shared strings with a regular list of regular strings:

```
typedef std::basic_string<
        char,
        std::char_traits<char>,
        shared_alloc<char>
    > shared_string;

std::list<shared_string, shared_alloc<shared_string> > SharedList;
std::list<std::string> TestList;
```

Not only will `operator==` fail to compile, but employing iterators and standard algorithms will not work either:

```
bool same = std::range_equal(SharedList.begin(), SharedList.end(),
                             TestList.begin(), TestList.end());
```

The types to which the iterators refer are not equality-compatible (`std::string` vs. `shared_string`). The interoperability barrier caused by the use of template implementation policies impedes the straightforward use of ***vocabulary types*** – ubiquitous types used throughout the internal interfaces of a program. For example, to declare a string, `s` using `MyAllocator` we would need to write

```
std::basic_string<char, std::char_traits<char>, MyAllocator<char> > s;
```

Many people find this hard to read, but the more important fact is that `s` is not an `std::string` object and cannot be used wherever `std::string` is expected. Similar problems exist for other common types like `std::vector<int>`. The use of a well-defined set of vocabulary types like `string` and `vector` lends simplicity and clarity to a piece of code. Unfortunately, their use hinders the effective use of STL-style allocators and vice-versa.

Finally, template code is much harder to test than non-template code. Templates do not produce executable machine code until instantiated. Since there are an unbounded number of possible instantiations for any given template, the number of test cases needed to ensure that every path is covered can grow by an order of magnitude for each template parameter. Subtle assumptions that the template writer makes about the template's parameters may not become apparent until someone instantiates the template with an innocent-looking, but not-quite-compatible parameter, long after the engineer who created the template has left the project.

Template implementation policies can be very useful when constructing mechanisms, as in the case of a function object (functor) type being used to specify an implementation policy for a standard algorithm template. Alexandrescu makes a compelling case for the use of template class policies in situations where instantiations are not expected to interoperate. However, template implementation policies are detrimental when used to control the memory allocation mechanisms of basic types that could otherwise interoperate.

4.4 Who's Got My Allocator?

When an allocator object is copied, the new object presumably compares equal to the original and thus must refer to the same memory resource as the original. The `rebind` template described earlier strengthens this requirement. Sometimes a single container will need to allocate memory for more than one object type. Rather than keep multiple allocators around, the container will typically keep one allocator member and rebind it each time a different allocator type is needed. For example, a `deque<T>` will typically need to allocate arrays of `T` and an array of pointer-to-`T`. If it has a member, `alloc`, of type `MyAlloc`<T>, then it may use the following statement to allocate an array of pointers:

```
typename MyAlloc::template rebind<T*>::other ptrAlloc(alloc);
ptrArray = ptrAlloc.allocate(n);
```

The first line creates a pointer allocator from the non-pointer allocator. The second line actually allocates the array. Presumably, `ptrAlloc` is a local variable that will go out of scope as soon the block exits. However, the memory allocated in the second statement needs to be freed eventually, say, using a similar sequence in the destructor:

```
typename MyAlloc::template rebind<T*>::other ptrAlloc(alloc);
ptrAlloc.deallocate(ptrArray);
```

In order for this allocation strategy to work, `ptrAlloc` must access the same memory resource in both code fragments. Thus, we must be able to assume that constructing one allocator from another produces the same results every time. If the allocator has per-instance state, then it would most likely be implemented as a small class containing a single pointer to some data structure that is shared across all allocator objects that are direct or indirect copies of one another. Thus, when considering the impact of per-instance allocators, it is useful to think of each allocator object as if it were a pointer.

In section 32.1 (container requirements) the 1998 standard says that two container objects, `c1` and `c2`, of the same type, `T`, compare equal if the following expression returns `true`:

```
c1.size() == c2.size() && std::equal(c1.begin(), c1.end(), c2.begin())
```

Notice that the allocator used to construct `c1` and `c2` is not mentioned. The allocator is, by implication, an implementation detail and is not part of each container's ***value***. This is entirely reasonable, but the standard is not consistent on this point. Although the assignment operation,

```
    c1 = c2;
```

does not copy c2's allocator to c1, the copy constructor,

```
    T c3(c2);
```

does copy c2's allocator to c3! This inconsistency plays havoc with the programmer's choice of allocator instance.

Consider the process of inserting elements into a vector of integer vectors. The inner vector type uses a per-instance custom allocator, which is constructed using an enumerated value. Let's start by creating two such allocators and two integer vectors:

```
    CustomAlloc<int> alloc1(SYSTEM_MEM);
    CustomAlloc<int> alloc2(LOCAL_MEM);

    typedef std::vector<int, CustomAlloc<int> > IntVecType;

    IntVecType v1(alloc1);
    IntVecType v2(alloc2);
```

Our first insertion into an empty vector of vectors is predictable:

```
    std::vector<IntVecType> vv;
    vv.push_back(v1);
```

The push_back operation will copy construct vv[0] from v1 causing vv[0] to use the same allocator as v1 (i.e. alloc1), which by itself can be problematic. The alloc1 object may be holding on to resources that you want to release, but you cannot release them until vv goes out of scope. If vv[0] is copied again, you can easily lose control over where the allocator is being used. The problem is made worse if we use an insert operation:

```
    vv.insert(vv.begin(), v2);
```

The new first element, vv[0], is a copy of v2, but what allocator does it use? The answer is not specified by the standard. Depending on the capacity of vv at the time of insert, the state of vv[0] may have been set by either copy construction or by assignment and may thus use a copy of alloc2 or of alloc1. We have lost control of our allocators and defeated the whole point of using allocators in the first place: to maintain explicit control over memory.

Some people see allocators as a way to improve the efficiency of memory management within their program and believe that per-instance allocators are not needed for this purpose. The theory is that, by selecting an allocator type that is tuned to the type objects being allocated, substantial performance improvements can be realized. Studies [Berger02] and personal experience have shown, however, that such ***per-class*** allocators produce little gain in performance over state-of-the art general-purpose allocators. The Berger study shows that consistent performance benefits are seen only when using region (or arena) allocators – allocators that use memory from within one or more large regions that allow all of the memory to be freed at once. Our own experience with large multi-threaded applications showed that thread-specific allocators can also provide dramatic performance benefits by reducing contention on the common heap (although not all multithreaded allocators have contention problems). Both arena allocators and thread-specific allocators rely on our ability to control what objects use the allocator and our ability to be able to determine reliably when an allocator is no longer in use. Because the standard allocator model does not give us sufficient control over per-instance allocators and because global allocators have limited utility, there seems to be little benefit to using allocators according to the standard model.

5 The Best of Both Worlds

5.1 Merging the Allocator Models

Our experience with BDE shows that per-instance allocators are much more useful than per-type allocators. The effort that has gone into solving issue 431 indicates that we are not the only ones who want to be able to use per-instance allocators. The challenge before us was to find a way to extend the STL allocator mechanism to support BDE-style allocation semantics while retaining compatibility with existing standard-compliant code. For our starting point, we chose an existing implementation of the C++ Standard Library with good compliance with the standard and excellent portability to all of our platforms. We then went about making changes to the library in such a way that it remained compliant with the 1998 standard, but also supported the Lakos allocator model.

5.1.1 Wrapping `bde::Allocator`

The obvious first step in bringing the STL and BDE allocator worlds together was to create an STL allocator template that was implemented as a simple wrapper around a `bde::Allocator` pointer. We debated whether to modify `std::allocator` or to create an entirely separate allocator template. In the end, the need for vocabulary types like `string` and `vector<int>` as well as our desire for the new allocator to interoperate with existing third-party libraries convinced us that modifying the implementation of `std::allocator` and retaining the type name was the preferred approach.

The resulting `std::allocator` template looks like the `MyAllocator` template described above. Its default constructor has the following signature:

```
    allocator(bde::Allocator *baseAllocator = 0);
```

Note that this constructor acts as a conversion operator, automatically converting a `bde::Allocator` pointer to a `std::allocator` object. The `bde::Allocator` pointer is stored in a member variable and the standard `allocate` and `deallocate` functions are implemented as simple pass-through calls to the corresponding (virtual) functions in `bde::Allocator`. If no pointer is specified, the constructor obtains a pointer to the new/delete allocator singleton, thus making the behavior identical to the standard behavior for code that is unaware of this change.

5.1.2 Modified Copy-Constructor Semantics

Because our modified `allocator` template now carries state, we must concern ourselves with the semantics of container copying. As described previously, the copy construction of a container should not cause the allocator to be copied, but we want control over the new container's allocator. We accomplish this objective by defining two constructors that can be used to copy each container (example is for `std::vector`.):

```
    vector(const vector& original, const allocator_type& alloc);
    vector(const vector& original);
```

The first constructor allows the caller to set the allocator explicitly. The second constructor uses `allocator_type()`, i.e., the default value of the allocator. Neither constructor copies the allocator from `original`. Unfortunately, this modification is technically incompatible with existing code that uses STL-style allocators. We remedied this incompatibility by specializing the second constructor such that `original.get_allocator()` is copied (as per the 1998 standard) for any old-style (non-

BDE) allocators. (See the proposal section for details on the traits-based declarations that make this compile-time selection of behaviors possible.)

Note that our modified `std::allocator` is considered a BDE-style allocator and thus triggers the new copy semantics. For existing (third-party) code that is ignorant of this enhancement, the behavior will not appear to have changed, since such code would never explicitly pass an allocator object to a container that uses `std::allocator` and would therefore always get the default `operator new`/`operator delete` behavior. Such code can also manipulate a container through a pointer or reference, oblivious to whether or not the container is using the default allocator:

```
void uintToString(std::string* s, unsigned i);  // existing 3rd party function
char buffer[200];      // Short-term memory for string
bde::BufferAllocator alloc(buffer);
std::string s(&alloc); // string will allocate memory from buffer
uintToString(&s, 15);  // uintToString doesn't care that s has a non-default allocator
```

In the example above, string `s` has allocator type `std::allocator<char>`. However, there is an automatic conversion from `bde::Allocator*` to `std::allocator<T>`, so the initialization of `s` is equivalent to

```
std::string s(std::allocator<char>(&alloc));
```

In this way, we are able to supply, at construction time, a BDE-style allocator for any string or container that was instantiated with the default allocator. The compile-time type of the string or container is unaffected.

5.1.3 Container and Contained Elements Using the Same Allocator

We then set about changing the containers so that they share their allocators with their contained elements using the same type traits system described previously. The new semantic was applied only if both the container and the contained elements were instantiated using the same new style allocator. For most container types, this transformation involved modifying only a few points in the logic where the item's copy constructor was being used. However, because of the way the library was originally written, we had to completely rewrite `vector` and `deque` in order to get this piece of functionality.

Ideally, any user-defined class that needs to allocate memory (directly or via a container member) should use an allocator. We made it a habit to put an optional `bde::Allocator*` argument at the end of the constructor arguments (including the copy constructor arguments) of any class that allocates memory. This constructor argument can be used to initialize the allocator for any STL container member. For example, a `Calendar` class might be declared like this:

```
class Calendar {
  private:
    std::vector<std::time_t> holidays;
    // ...
  public:
    // Default constructor:
    explicit Calendar(bde::Allocator* basicAllocator = 0);

    // Copy constructor:
    Calendar(const Calendar&  other,
             bde::Allocator* basicAllocator = 0);

    // Another constructor
    Calendar(std::time_t startDate, std::time_t endDate,
             bde::Allocator* basicAllocator = 0);
};
```

The default constructor would pass the allocator through to its container member(s):

```
Calendar::Calendar(bde::Allocator* basicAllocator)
  : holidays(basicAllocator)
{
    // ...
}
```

This logic is the same for the other constructors. We take advantage of the automatic conversion from `bde::Allocator*` to `std::allocator<size_t>` and the fact that this conversion automatically uses the new/delete allocator if it is passed a null pointer.

5.2 Move-Semantic Operations

5.2.1 Copying Values vs. Moving Objects

We say that a type has ***value semantics*** if it has a well-defined notion of value, typically defined operationally by `operator==`, that is copied by the copy-constructor and assignment operator. STL container types are value-semantic types, as are strings, built-in types, pointers and enumeration types. Two containers compare equal if each of their elements compare equal – the containers' allocators are not examined by `operator==` because the allocators are not part of the containers' values.

Value semantic types are closely associated with copy-semantic operations – operations that copy the value of an object without modifying the object being copied. However, the C++ language and library define certain operations which have what we call ***move-semantics***. A ***move-semantic operation*** is an operation on an object that conceptually moves the entire object, including its non-value attributes, to a new location. Examples of move-semantic operations are `std::swap`, copy-avoidance optimizations (including the RVO), and some uses of the proposed rvalue reference [N1690]. After a move operation, an object's ***identity*** appears to have moved to a new address. Most uses of move-semantic operations are optimizations to reduce the number of copy operations.

Move-semantic operations do not invalidate our allocator principles if our notion of "object" is flexible enough to include the possibility that the object may be moved. We must be extra careful, however, because the optimizations provided by move-semantic operations makes it tempting to apply them where they could cause unintended effects. (Note that this issue is not limited to allocators. There are other examples of non-value attributes that are not expected to be copied when the value is copied: Imagine a mutex member of a thread-safe object.) The purpose of this section is to explore the interaction between move semantics and non-value attributes (especially allocators) and propose some algorithms and practices that can be applied to improve the chances of writing correct code.

5.2.2 Swap

The 1998 standard requires that `swap`, when invoked on standard containers, must be a constant time operation and must never throw. This is accomplished by re-assigning pointers within the container data structure, without actually copying any elements. This becomes a problem when the containers being swapped have different allocators, since it would separate allocated memory from the allocator needed to de-allocate it, as described in issue number 431 of the Library Active Issues List [Issue431].

Some favor resolving issue 431 by swapping the allocators along with the values when swap is called to exchange the contents of two containers with unequal allocators [N1599]. This resolution allows swap to remain an O(1) operation, retains the nothrow guarantee, and prevents invalidation of iterators and references. An unstated consequence of this change is that swap becomes a move-semantic operation for standard containers – the containers being swapped effectively ***trade places*** rather than just trade values. This is reasonable behavior for most current uses of swap. For example, it is probably appropriate for algorithms like `std::sort` and `std::reverse`, to conceptually move elements around, and the performance benefit can be substantial.

Other uses of swap may require extra care. Swap is often used to provide a strong exception-safety guarantee. For example, the following function uses swap to atomically replace the value of a string:

```
void uintToString(std::string* s, unsigned i)
{
    std::string temp;
    do {
        temp.insert(0, 1, i % 10 + '0'); // might throw
        i /= 10;
    } while (i != 0);
    s->swap(temp); // never throws
}
```

For this use of swap, the main criteria is that swap must not throw an exception. However, the swap operation in the code above might change the allocator type used by `*s`, with undesirable consequences. This problem is easily remedied by carefully choosing the allocator when constructing the temporary variable:

```
void uintToString(std::string* s, unsigned i)
{
    std::string temp(s->get_allocator());
    do {
        temp.insert(0, 1, i % 10 + '0');
        i /= 10;
    } while (i != 0);
    s->swap(temp);
}
```

With the above change, the swap operation is guaranteed to be invoked on two objects with the same allocator, making the distinction between move semantics vs. copy semantics mute. Explicitly setting the allocator of the object to be swapped is good practice, even if this proposal is not accepted into the standard.

The most dangerous (and, fortunately, least common) use of swap is to exchange the values of two unrelated objects. The user in this case would almost certainly be adversely affected by the allocator change caused a move-semantic swap. Rather than leave this as a trap for the unsuspecting novice, we propose to create a second algorithm, `swap_value` which, by default, is implemented as the standard three copy operations (temp = a, a = b, b = temp). The `swap_value` algorithm has neither the complexity guarantee nor the nothrow guarantee of swap. (Note that swap is only guaranteed O(1) and nothrow for standard containers and other classes that make this guarantee explicitly.) Trivial though it is, there are good reasons to make `swap_value` a part of the standard:

- Having `std::swap_value` along with (and alphabetically adjacent to) `std::swap` should cause the casual user to ponder a moment before selecting the algorithm that is appropriate to the situation.

- Containers can optimize `swap_value` to be as fast as `swap` in the case of equal allocators, and to use only two copies rather than three in the case of unequal allocators:

```
template <typename T, typename Alloc>
void swap_value(vector<T,Alloc>& a, vector<T,Alloc>& b) {
    if (a.get_allocator() == b.get_allocator()) {
        a.swap(b);
    }
    else {
        vector<T,Alloc> temp(a, b.get_allocator());
        a = b;
        b.swap(temp);  // Avoid third copy operation
    }
}
```

If one desires a copy-semantic swap operation with the strong exception safety guarantee, this can be achieved at the cost of additional memory using the following algorithm ("Halpern's Algorithm"):

```
template <typename T, typename Alloc>
void swap_value_atomic(vector<T,Alloc>& a, vector<T,Alloc>& b) {
    // Exception-safe swap of two vectors using Halpern's Algorithm
    if (a.get_allocator() == b.get_allocator()) {
        a.swap(b);
    }
    else {
        // Unequal allocators.  Copy each vector using the other
        // one's allocator. An exception will leave both a and b
        // unmodified.
        vector<T,Alloc> temp_a(a, b.get_allocator()); // might throw
        vector<T,Alloc> temp_b(b, a.get_allocator()); // might throw
        a.swap(temp_b); // nothrow
        b.swap(temp_a); // nothrow
    }
}
```

At this time, we are not proposing `swap_value_atomic` for inclusion in the standard.

5.2.3 Elided Copy Constructors

An implementation is permitted avoid calling a copy constructor for making a copy of a temporary variable or for creating a temporary variable from the return value of a function (the return-value optimization). In each of these cases, the original object and the would-be copy are treated as the same object. These optimizations have the qualities of move-semantic operations; in particular, the allocator and other non-value attributes are determined not by the new object's copy constructor, but by the constructor of a conceptually different object elsewhere in the program. The presence of these optimizations are theoretically problematic, but the problems can be easily avoided with simple programming conventions. To illustrate the problem, consider this function:

```
std::string tellMe(int time) {
    char buffer[200];  // temporary buffer
    buffer_allocator alloc(buffer, 200); // temporary allocator
    std::string temp(&alloc);  // uses temporary allocator
    temp = "I'm telling you for the ";
    switch (time) {
        case 1: temp += "first";  break;
        case 2: temp += "second"; break;
        case 3: temp += "third";  break;
    }
    temp += " time!";
    return temp;
}

std::string told = tellMe(2);
```

Section 12.8, paragraph 15 of the standard says that a copy constructor does not need to be invoked to copy `temp` to the return value of `tellMe`, nor does a copy constructor need to be invoked to copy the return of `tellMe` into `told`. Instead, `temp` can be constructed directly into `told`, avoiding two copies. Unfortunately, this means that the allocator for `told` is determined not by the constructor for `told`, but by the constructor for `temp`. To make matters worse, `temp` uses a temporary allocator that goes out of scope before `told` is destroyed. The solution to this problem is to simply get into the habit of always using the default allocator instance for any returned value, either by constructing it that way:

```
std::string temp; // use default allocator instance
```

or by an explicitly copying it (using the extended copy constructor) on return:

```
return std::string(temp, std::allocator());
```

5.2.4 Rvalue References

A proposal for rvalue references [N1690] provides a way to explicitly modify temporary variables, often by “pilfering” their contents using `swap`. Once again, care must be taken to ensure that the desired allocator semantics are preserved. For example, the optimization enabled by the rvalue reference could be disabled for unequal allocators:

```
void myclass::setValue(std::string&& v) {
    if (v.get_allocator() == value.get_allocator())
        value.swap(v);  // pilfer temporary variable
    else
        value = v;      // slow copy, preserve value's allocator
}
```

This logic can also be incorporated into an rvalue extended copy constructor:

```
template <typename T, typename Alloc>
X::X(X&& other, allocator_type alloc) {
    if (alloc == other.get_allocator) {
        // pilfer contents of other
    else
        // copy the contents of other
}
```

5.3 Reaping the Benefits

The BDE system of per-instance allocators has had a profound impact on the way we manage memory. The original STL allocators were used only in very specialized corners of an application, where per-type allocators did not pose a problem. The new allocators, conversely, are used throughout our code to give us more control over memory use. Some of the ways we use allocators are:

- In a very large system, we share objects across processes simply and efficiently using a shared-memory allocator. Because containers propagate their allocators to their contained elements, only the root of a complex data structure needs special treatment in order to be shared.

- In a module that reads data from an external source, we use a limit allocator to prevent denial-of-service attacks.

- In a messaging module, we build up a complex container-within-container data structure using a very fast arena allocator (`allocate` simply returns the next sequential memory block, `deallocate` does nothing). When the message is complete and has been transmitted, we delete the allocator, releasing all memory at once, without ever calling a destructor. (This approach requires that the objects in question not acquire any resources except for memory and that they allocate all of their memory using the allocator. The destructors must not have any side effects.)

- Hundreds of unit-test drivers use a test allocator to ensure that the components under test neither leak memory nor use it extravagantly. The test drivers also use the test allocator to simulate out-of-memory conditions in order to prove that the components under test are robust in the face of exceptions.

6 Formal Proposal

Everything I've written up to this point constitutes the motivation for a proposed change to the containers and allocators section in the emerging revision of the C++ standard. What follows is a preliminary draft of a formal proposal for enhancements to the C++ standard library for C++0x. This proposal basically promotes the BDE modifications to the STL allocator mechanism (with names and other details changed to be more standard-like). The BDE project, as described above, provides an example of an ***existing implementation*** and invaluable source of experience.

1. Add a new abstract class `allocator_implementation` to the <memory> header and a concrete derived class `newdelete_allocator_implementation` as follows:

```
class allocator_implementation {
  public:
    typedef size_t size_type;

    virtual void* allocate(size_type n, void* hint = 0) = 0;
    virtual void deallocate(void* p) = 0;
    virtual ~allocator_implementation();
};

class newdelete_allocator_implementation
    : public allocator_implementation {
  public:
    static newdelete_allocator_implementation* singleton();

    void* allocate(size_type n, void* hint = 0);
    void deallocate(void* p);
};
```

   To make these classes easier to use, overload placement new and `delete` operators for the base class:

```
Void* operator new(allocator_implementation& a, size_t bytes);
Void operator delete(allocator_implementation a, void* p);
```

   Effects: `operator new` returns `a->allocate(bytes, 0)`.
   `operator delete` calls `a->deallocate(p)`.

   Note: The placement-style `operator delete` described here is called only when an exception is thrown from a constructor while using the corresponding placement-style `operator new`.

Our proposal does not present a one-step method for destroying and de-allocating an object that was successfully created using the `operator new` described here.

2. Modify the constructor and add members to the default allocator template as follows:

```
template <typename TYPE>
class allocator {
    allocator_implementation* imp; // exposition only
  public:
    allocator(allocator_implementation* i = 0);
    allocator_implementation get_implementation() const;
    void swap(allocator& other) throw();
    // rest of template remains the same
};
```

   Constructor effects: If ***i*** is non-zero, initialize ***imp*** to ***i***, otherwise initialize ***imp*** to `newdelete_allocator_implementation::singleton()`.

   Allocate effects: Returns `imp->allocate(n * sizeof(value_type))`.

   Deallocate effects: Calls `imp->deallocate(p)`.

   Swap effects: `swap(this->imp, other.imp)`.

3. Add to the allocator requirements that for any allocator type, `A`, there must be a function `swap(A&, A&) throw()` to exchange the mechanisms of two allocators such that after the call each allocator controls the memory previously controlled by the other. Add the following definition to `<memory>`:

```
template <typename Type>
void swap(allocator<Type>& a, allocator<Type>& b) throw();
```

   Effects: `a.swap(b)`.

4. Declare two traits classes (see TR1 traits) as follows. [***The library working group may choose a better name than new_allocator to describe the Lakos allocator model.***] These traits allow container classes to share their allocators with their contained elements automatically:

```
template <typename Alloc> struct is_new_allocator : false_type;
template <typename Type> struct uses_new_allocator : false_type;
```

   With partial specializations as follows:

```
template <typename Type>
struct is_new_allocator<allocator<Type> > : true_type { };

template <typename Type, typename Alloc>
struct uses_new_allocator<vector<Type, Alloc> >
    : is_new_allocator<Alloc> { };
```

   Specialize similarly for other container and string class templates.

   In general, a class should specialize `uses_new_allocator` to evaluate to `true_type` if it uses an allocator for which `is_new_allocator<allocator_type>::value` is true and if it has a copy constructor that takes an (optional) allocator. A class should specialize `is_new_allocator` if it is an allocator that is intended to be used with the semantics described in this paper (i.e., propagated to contained elements and not copied when container is copied).

Note: If the concepts proposal [N1758] is accepted into the standard, these traits could be expressed as concepts instead.

5. For each standard container, container adaptor, or string template class, `Container`, add an "extended copy constructor":

   ```
   Container(const Container& c, const allocator_type& alloc);
   ```

   The allocator, `alloc`, is used by the container for internal memory allocations.

6. For each standard container, container adaptor, or string template class, `Container`, change the meaning of the normal copy constructor (`Container(const Container& c)`) as follows:'

   If `is_new_allocator<allocator_type>::value` is true, then construct the container as though it were constructed with `Container(c, allocator_type())`. I.e., the allocator from `c` is not copied.

   Otherwise, construct the container as though it were constructed with `Container(c, c.get_allocator())`. This behavior conforms to the 1998 standard.

7. For each standard container or container adaptor, `Container`, add an "extended default constructor":

   ```
   explicit Container(const allocator_type& a);
   ```

   (`vector`, `deque`, and `list` already have a constructor like this.) Since this proposal is intended to make per-instance allocators more useful, this constructor is intended to make them more convenient as well. In addition, for `stack`, `queue` and other container adaptors, it is necessary to provide a way to specify an allocator in the first place, since the allocator is not necessarily copied from the underlying container.

8. Each non-container class defined in the standard, `T`, that is copy-constructible and assignable and which allocates memory (e.g., `TR1::function`) must use an allocator. The trait `uses_new_allocator<T>::value` must be true. The semantics of `T`'s copy constructor must match the description in item 6. `T` must also have an ***extended copy constructor*** as described in item 5. If `T` has a default constructor, then it must also have an ***extended default constructor*** as described in item 7.

9. For each standard container, `Container`, enhance the meaning of `insert, push_back`, and other functions that construct new elements within the container as follows:

   If `uses_new_allocator<value_type>::value` is true ***and if*** `is_new_allocator<allocator_type>::value` is true ***and if*** `allocator_type` is convertible to `value_type::allocator_type`, ***then*** construct a new element with value `v` by calling the constructor `value_type(v, get_allocator())`. This logic propagates the container's allocator to each contained element.

   Otherwise (if the above conditions are not all true), then construct the new element with `value_type(v)`. This behavior conforms to the original 1998 standard.

10. Enhance `pair<T1, T2>` in a manner similar to containers:

If uses_new_allocator<T1>::value is true ***or*** uses_new_allocator<T2>::value is true, then uses_new_allocator<pair<T1, T2> >::value shall also be true and the following members will be added to the pair class (in addition to the members already defined in the current standard):

```
typedef see below allocator_type;
pair(const T1& v1, const T2& v2, const allocator_type& alloc);
pair(const pair& p, const allocator_type& alloc);
```

For both of these constructors, alloc is used as the second argument of the extended copy constructor for first or second or both. The allocator type is either typename T1::allocator_type::rebind<void>::other or typename T2::allocator_type::rebind<void>::other, depending on which parameter has the uses_new_allocator trait. If both T1 and T2 have the trait, then T1 is chosen. If both have the trait but T1::allocator_type is not convertible to T2::allocator_type then the program is ill-formed.

Note: this change is important for implementing the allocator semantics in map and multimap.

11. Add overloaded versions of the uninitialized_copy, uninitialized_fill, and uninitialized_fill_n algorithms to make constructing contained elements easier:

```
template <typename InputIterator, typename ForwardIterator,
          typename Allocator>
uninitialized_copy(InputIterator first, InputIterator last,
                   Forwarditerator result, const Allocator& alloc);

template <typename ForwardIterator, typename T, typename Allocator>
uninitialized_fill(ForwardIterator first, ForwardIterator last,
                   const T& x, const Allocator& alloc);

template <typename ForwardIterator, typename Size, typename T,
          typename Allocator>
uninitialized_fill_n(ForwardIterator first, Size n,
                     const T& x, const Allocator& alloc);
```

In all of these functions, the meaning is the same as the corresponding algorithm without the allocator argument except that ***if*** uses_new_allocator<typename ForwardIterator::value_type>::value is true ***and*** Alloc is convertible to typename ForwardIterator::value_type::allocator_type ***then*** ForwardIterator::value_type is constructed using allocator alloc.

12. Add an allocator argument to the constructors of stringbuf, stringstream, istringstream, and ostringstream. (If any replacement is created for strstream, it, too, must have an allocator argument.)

13. Add a mutating algorithm:

```
template <typename T> void swap_value(T& a, T& b);
```

Effects: temp = a; a = b; b = temp

14. swap_value shall be overloaded for each standard container and string class template:

```
template <typename T> void swap_value(Container<T>& a, Container<T>& b);
```

Effects: exchange the elements, but not the allocators of a and b.

Complexity: constant time if `a.get_allocator() == b.get_allocator()`, `a.size() + b.size()` copy operations on `T` (each element is copied once) if `a.get_allocator() != b.get_allocator()`.

Throws: nothing unless copy operation on `a` or `b` or one of their elements throw. `a` and `b` will have indeterminate (but valid) values in case an exception is thrown

[Note: we could also add a `swap_value_atomic(Container&,Container&)` function that would have no default declaration but would be defined for each standard container type. It would add the guarantee that `a` and `b` would be unmodified in case of exception (e.g. using Halpern's algorithm).]

7 Conclusion

The 1998 C++ standard defines an allocator model in which allocation policy is specified as a template parameter to container class templates. We found the standard's allocator model deficient in a number of respects: container types instantiated with one allocator do not interoperate with otherwise identical container types instantiated with different allocators. Even when this is not an issue, the semantics of copying allocators causes one to lose control over his/her allocator objects, making per-instance allocators impractical.

The BDE project uses a powerful model, developed by John Lakos, of per-instance allocators that cleanly separate a container's memory allocation policy from its type and value. BDE containers automatically share their allocator with their contained elements, producing a single memory allocation domain for each complex object. This mechanism proved very powerful, easy to use and easy to extend. We were able to use it extensively to achieve data sharing, increase efficiency, and facilitate testing.

Merging Lakos's allocator model into an existing implementation of the C++ standard library, we were able to get the best of both worlds. Our new STL continues to comply with the 1998 standard, but we were able to get the benefits of per-instance allocators on which we had come to rely. In addition, we implemented a solution to an outstanding standard library issue 431 (Swapping containers with unequal allocators) that is consistent with our allocator philosophy. Our experience provides an ***existing implementation*** and points the way to a general approach that can be incorporated into the emerging revision of the standard.

8 Bibliography

[Alexandrescu01] Alexandrescu, Andrei, ***Modern C++ Design***, Addison-Wesley 2001

[Berger02] Berger, Emery, McKinley, Kathryn and Zorn, Bengamin, ***Reconsidering Custom Memory Allocation***, ACM OOPSLA'02 2002. http://www.cs.umass.edu/~emery/pubs/berger-oopsla2002.pdf

[Issue226] Abrahams, Dave (submitter), ***C++ Standard Library Active Issues List: User supplied specializations or overloads of namespace std function templates.*** ISO/IEC 2000. http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-defects.html#226

[Issue431] Austern, Matt (submitter)***, C++ Standard Library Active Issues List: Swapping containers with unequal allocators***. ISO/IEC 2003.
http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#431

[Lakos96] Lakos, John, ***Large-Scale C++ Software Design***, Addison-Wesley 1996

[LoRusso01] Lo Russo, Graziano, ***An Interview with A. Stepanov***, Edizioni Infomedia srl., 2001
http://www.stlport.org/resources/StepanovUSA.html

[N1599] Hinnant, Howard, ***Issue 431: Swapping containers with unequal allocators*** JTC1/SC22/WG21 N1599=04-0039, 2004
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1599.html

[N1690] Hinnant, Howard, Abrahams, Dave and Dimov, Peter, ***A Proposal to Add Rvalue Reference to the C+ Language***. ISO/IEC JTC1/SC22/WG21 N1690=04-0130, 2004.
http://www.open-std.org/jtc1/sc22/wg21/docs/2004/n1690.html

[N1758] Jeremy Siek et al, ***Concepts for C++0x***. ISO/IEC JTC1/SC22/WG21 N1758=05-0018, 2005.
http://www.open-std.org/jtc1/sc22/wg21/docs/2004/n1690.html

[Maddock01] Maddock, John, Cleary, Steve, et al., ***Type Traits***, boost.org 2001.
http://www.boost.org/libs/type_traits/index.html

[Stepanov95] Stepanov, Alexander and Lee, Meng, ***HP Labs Technical Reports: The Standard Template Library***, Hewlett-Packard Laboratories 1995.
http://www.hpl.hp.com/techreports/95/HPL-95-11.html

[TR1] Austern, Matt (editor), ***Proposed Draft Technical Report on C++ Library Extensions***, ISO/IEC 2005.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1745.pdf