Doc Number: N2486=07-0356
Date: 2007-12-05
Reply To: Lance Diduck
lancediduck .a. nyc…rr…com
lance.diduck .a. ubs…com

Alternative Allocators and Standard Containers

A look at C++ alternative allocators specification.

Overview

A common complaint among standard container users is performance. Custom allocators can dramatically improve the situation, but the current way to parameterize allocators leaves a lot to be desired. Indeed, many users find it easier to just implement containers themselves, than figure out how to make an allocator work with their native library. Worse, custom container authors have almost no clue on how to fully take advantage of allocators in their own offerings. The standard offers no guidance.

The language of the standard is over-constrained, under-constrained, or simply absent in terms of just what kinds of allocators are supported, and how they interact with the containers.

This paper presents an allocator specification and implementation that

- Reduces Boilerplate – this is perhaps the single biggest impediment for a user.
- Is backward compatible – the allocator code works with current container implementations as-is, and the container modifications work with the current std::allocator as-is
- Does not promote a particular style of programming. The current intent of the standard is considered the preferred style (but certainly not the only style).
- No pet ideas –almost every recommendation here has been heard elsewhere.
- Reflects what is available now with most current library implementations and compilers
- Does not undo any of the functionality envisioned in the original allocator design
- Accommodate virtually all the current standard as written (e.g. complexity guarantees, function signatures) Most additional language largely reflects current implementations. Eliminates weasel wording
- Keep simple and easy to teach (Compared to allocators now, anything is simpler)

This paper does not address specific changes to the wording of the standard. This is easy to remedy in a revision, if desired.

To accommodate most current compilers, all code presented is C++03, without template template parameters. These and the C++0x enhancements[1] can be added, which would get rid of even more boilerplate.

The containers prefixed with "std::" used in the examples are either native (tested using Visual Studio 2003 /2005, and/or gcc 3.4 and 4.1, but see this for others when only alternative allocate/deallocate is wanted) . The containers without the std:: prefix are modified versions of

[1] For example, perfect forwarders in construct()

stdcxx[2] containers. The modifications are largely to accommodate this papers wording on construct/destroy, alternative pointers, and copy assigning the allocator. These modifications still default to unmodified std::allocator. Modified container code is not shown[3].

Motivation

These are real life use cases, used with commonly available compilers and standard libraries. No thought experiments.

A performance check

We will start with a performance check, to find out why C++ users bother with allocators, and want the “weasel wording” out:
An application was (properly) written using out of the box native containers and allocators, and profiling indicates a bottleneck when inserting and erasing list<string>.
We make a small program that mimics the bottleneck, and try out different things. This is run on four concurrent threads, and the times are averaged together. Each test uncomments one of the typedefs:

```
//What the application uses
//  typedef std::list<std::string> test_list;
//Base line test uses a vector
//  typedef std::vector<std::string> test_list;
//List with full allocator support, with defaults
//  typedef FullAlloc::list<std::string> test_list;
//List with tuned allocator, in a wrapper
//  typedef ObjectList<std::string,2500> test_list;
void* perftest(void*){
   unsigned long long start=get_timestamp();
   //application creates list of strings, and repeatedly
   // fill it up, manipulates the string, and then clears the list
   // profiling indicates bottleneck in adding and clearing strings
   test_list mytest;
   for (std::size_t i=25000;i;--i){
         for(unsigned j=2500;j>30;--j){
               mytest.resize(mytest.size()+1);//add new string
               mytest.back().resize(j%50+20,char(j));//a value
         }
         //app manipulates string, then clears everything
         mytest.clear();
   }
   unsigned long long finish=get_timestamp();
   return reinterpret_cast<void*>(static_cast<unsigned>(
              (finish-start)/100000));
}
```

The Base Line test runs at 1 unit (avg. 39.0 seconds on my box). `std::vector` is a very simple stateful allocator and naturally has no thread contention in this case. For this

[2] Chosen primarily for licensing, portability and current feature set. I also had a modified version of Dinkum containers used for testing, but these were abandoned due to licensing constraints. The Dinkum alternative allocator support is comparable (but not identical) to stdcxx.
[3] It is straightforward to make the modifications to any library that supports the optional stateful allocator argument.

particular benchmark it represents the fastest achievable time – so when we know when to quit trying.
The Native List test uses an average of 150% of vectors time (i.e. half again slower). The native std library does not fully support alternative allocators (like most, it is missing support for pointer and accessors). Part of the overhead is the pointer manipulations inherent in list, but certainly not an additional 50%.
“FullAlloc::list” is a slightly modified version of a list from a major STL provider, such that it *fully* supports alternative allocators (pointers, accessors, and allocations). Using defaults, it also takes average of 150% of vectors time – demonstrating that full allocator support does not add any time overhead, and is backwards compatible with the native std::allocator. Any differences are more likely due to what the compiler can optimize (see EASTL) rather than the feature support for allocators.
Now we make the tuned version: ObjectList is a convenience wrapper that inherits from FullAlloc::list, which stuffs in a “stack memory” allocator that maintains a free list of equal size blocks, *and* allows for caching live objects in lieu of raw memory. We make enough room for 2500 objects. Since the high water mark is reached early and rarely exceeded, an unbounded version would perform just as well. I just happen to have the “stack” version in my toolbox.
The buffer only method takes 107% of the time. If we add object caching to that buffer, then this becomes 108%. I figure that the compiler/native strings have some sort of tuning going on of their own, so I stay out of the way and kill object cache idea.[4] Part of the extra time is just for the pointer manipulations, so gains after this are harder to come by, and I should focus my attention elsewhere. I can easily extend the “stack” allocator to an unbounded version, and I am done.
If this had been rather list<int>, then the “ObjectList” version runs in 9% of the time that plain list<int> (either std:: or FullAlloc::) takes. But still 6 times slower than vector<int>.

In the end I have all the guarantees of a list, but at a fraction of the cost. And I know this is close to the best time I can get, because I can easily try out a number of different strategies. Total time, about 4 hours work.

What did we learn?

Foremost, that we need to match the allocator *behavior* to the specific application behavior. This point seems forgotten and relearned quite often, since perhaps allocators – even after a decade of use-- are commonly seen as being somehow divorced from the subsequent use of that memory. In other words, the typical viewpoint is that allocators consider only the *structure* of the type in question, and not its *use*.
This is perhaps because we all spend our formative years programming in languages where this is indeed the case. And even those few languages where the application programmer can select among allocators, most programmers only consider the structure of the type being allocated.
However, C++ uses behavioral typing in addition to structural typing, which makes it easy to select algorithms based on a data structures behavior. Indeed, this is the cornerstone of the STL.

[4] To contrive an example that demonstrates better performance using object caches would require far more time and space that this paper allows

We just saw that a programmer can use the type system to easily select and correctly apply various behavioral allocation schemes, and get a substantial performance increase. (But as always, premature optimization is the root of all evil[5].)
As we will see, the standard containers themselves only have a few behaviors, but even here there is a wealth of allocation options. The trouble is that nowhere in the standard does it specify just what container exhibits what allocation behavior. And once you do discover it, actually getting a container to portably use an alternative allocator, well, that exercise will make a C++ guru out of anyone. And it shouldn't.

A few ways to apply alternative allocators

Make Node Based Containers Fast

This is the classic application. There are many examples in use.

Thread Local Storage / Object Specific Allocator

vector and string have their own private heaps, why not any container? Private heaps mitigate the need for locking both the container AND the stateless allocator in MT programs, plus can keep objects packed close together. This is easy to implement using a stateful memory pool and a node container. The memory pool keeps allocating blocks that are later reclaimed by the system according to some policy, typically not until the container is destructed, just like vector / string. The swap trick works as well, if the container swaps allocators.
This is most prevalent use case I've seen of stateful alternative allocators. They are mostly used in multithreaded programming, to take advantage of the fact that the container object access must be serialized anyway, so why pay for two locks? In many cases the containers can be partitioned so that no locking is required.
The performance impact over std::allocator can be 20-50 times, especially in cases that one thread allocates, and another thread deallocates.
Support : every major STL provider

Reentrant Scratchpad

The use case that people first try their hand with stateful allocators is a container that uses stack memory. These are useful for containers that are local to a function.
The problem of course is knowing just how much memory you are going to need before hand, but otherwise the allocator is simple and easy to build.

```
void foo(){
   vector<int,stackallcator<int,1000> > mystuff;
};
```

Depending on just how many reallocations you are going to make mystuff do, this can work. It is brittle, but sometimes the performance gain is worth it.

Support : every major STL provider

[5] C.A.R. Hoare

Cache Coherent

It is helpful to locate the controlled sequence in memory adjacent to the container bookkeeping information, and that the controlled sequence be packed together as close as possible. This is similar to the “reentrant scratchpad” except that memory is recycled, resulting in a far less brittle design.
In practical terms, it is something like

```
template<class T,size_t N>
struct CCList:std::list<T,statefulalloc>{
    aligned_storage< <N+1*(sizeof(T)+nodeoverhead)> buf;
    CCList():std::list<T,statefulalloc>(buf,sizeof(buf)){}
//accessors
//max_size==N or more
};
```

where “statefulalloc” dices up buf in a freelist of ~N blocks. It is a simple matter to make this unbounded, where only the first N elements are packed.
Current Support : every major STL provider

Object Pool

A common problem in high performance application is: removing an expensive-to-construct element from the controlled sequence, only to immediately insert another new value. It is often the case that copy-assign would be faster – but for things like map and set, I simply can’t copy over existing elements in general.
Object Pool keeps a collection of all objects deallocated. construct() is overridden to do assignment (either by operator= or another class specific method), destroy() takes care of calling the element’s clear() or dispose() function or whatever. allocate() returns a already constructed object, either from the free list or newly created. All objects are destructed when the allocator is destructed.
Given a suitable implementation, deque makes a good way to implement object pools for container requiring random access iterators. For others we just use a free list.
And just Like Object Specific Allocator, this can be made stateful to avoid multiple locks.
Current Support : A few providers, depending on the container

Relocatable Container

“Relocatable” means that the contents of the memory do not depend their location in memory. This is a very important concept both in automatic memory garbage collection, and in interprocess IO. The former case is beyond the scope of what a *standard* container is capable of[6], but in the latter case is easily handled. Using a specialty pointer that stores offsets, rather than the raw pointer value, I can make a container capable of being streamed from one process to the next (of course, each process must be compiled using the same compiler settings, which is typical in high performance IPC). More likely, I am placing this container in Shared Memory. Shared Memory Transport is, after all, the fastest known IPC method. Likewise, such a container can easily be written to disk without having to transform it to some other format.

[6] The generally accepted rule that &*container.begin() yields a pointer to real memory effectively rules out any hope of using mark-sweep or copy collector GC methods inside a standard container.

This requires a suitable pointer definition, plus a relocatable stateful allocator (no polymorphic allocators!!!).
For example
```
template<class T,size_t N>
struct RelocList:std::list<T,relocstatefulalloc>{
   aligned_storage< <N*(sizeof(T)+ factor) > buf;
   RelocList ():std::list<T, relocstatefulalloc >(buf,sizeof(buf)){}
//
//max_size()==N (bounded)
};
```
With the proviso that T is also relocatable, RelocList is suitable for placing in shared memory, sending via a socket, placing in a file, fast copies (by memcpy) etc.

Current Support: A few providers, depending on the container

Clarification of Terms

Much of the present confusion is that there is no clear description of allocator concepts nor their programmability. Here is a possible taxonomy of allocators *from the perspective of the container*. It is divided into three kinds – the Storage, the Manager, and the Adapter.
Later there is code that ties all this together.

AllocatorStorage

This is the resource under consideration, almost always “raw memory.” It represents all the std::allocator typedefs, plus construct(), and destroy().

Table 1 Concepts of AllocatorStorage

	Refinement of	
Bounded	None	
Unbounded	Bounded	
Relocatable	Bounded	

Table 2 Programmability of AllocatorStorage

	Static (empty)	Nonstatic
DefaultConstructible	Yes	No
Swappable	Yes	No
MoveConstructible	Yes	Yes
MoveAssignable	Yes	Yes
CopyConstructible	Yes	No
CopyAssignable	Yes	No
EqualityComparable	Yes	always false

Examples

a pointer to char buf[100]; (bounded)
a region of memory acquired via shmget (bounded, relocatable)
memory is DMA addressable

region with processor affinity (NUMA)
region is never swapped to disk (i.e. `mlockall)(unbounded)`
memory obtained via brk() (malloc/free)

Discussion

AllocatorStorage represents the resource (almost always untyped raw memory) and where we find the specifications for alternative pointers and accessors.
Bounded represents a quantity defined by the application.
Unbounded means “limited only by system resources”
Relocatable is a specific requirement for types constructed using this memory, such that memcpy is a valid way to copy construct an object.
For instance, when the AllocatorStorage is Relocatable, then an alternative pointer that maintains offsets is desirable. Also, the AllocatorStorage can represent special regions of memory (e.g. DMA addressable) or memory under control by an AllocatorManager that cannot tolerate raw pointers (i.e. GC).
AllocatorStorage is too platform specific to specify in detail. For example, it is conceivable to make an AllocatorStorage that represents a XML DOM Level 3 binding, for use with a specialty container that modeled a Sequence. Unbounded, Bounded, and Relocatable are concepts that have broad usage however.

AllocatorManager

This is the representation of allocate(), deallocate(), operator==(), and max_size(). It has the semantics of a typical resource manager.
Each AllocatorStorage has one and only one AllocatorManager. An AllocatorManager can handle one or more AllocatorStorage (all of the same type).

Table 3 Concepts of AllocatorManager

	Refinement of	
Trivial	None	
FixedSize	Trivial	Works with class operator new
VariableSize	FixedSize	
Contiguous	VariableSize	Works with operator new[]
NonDeterministicReclaim	*FixedSize*	
ExpandInPlace	*Trivial*	
Hybrid	*VariableSize*	

Table 4 Programmability of AllocatorManager

	Static Case	nonstatic Case (in general)
DefaultConstructible	Yes	No
Swappable	Yes	Yes
MoveConstructible	Yes	Yes
MoveAssignable	Yes	Yes
CopyConstructible	Yes	No

CopyAssignable	Yes	No
EqualityComparable	always true	always false

Examples:

free list of fixed size blocks from AllocatorStorage (FixedSize)
Object Pool -- free list of live objects (FixedSize or VariableSize)
a pointer that gets "bumped" on each allocation (VariableSize)
The functions malloc/free/realloc (Contiguous + ExpandInPlace)
The POSIX functions mmap/mfree (Trivial)
The Linux functions mmap/mfree/mrealloc (Trivial + ExpandInPlace)
BDW / Hazard pointer (NonDeterministicReclaim)

Discussion

There are of course many more ways to manage memory than listed here. However, this is from the point of view of "what does a standard container expect" and not "what is possible"
Trivial is a manager that owns one AllocatorStorage, and allocate() simply returns the start address held by the storage, and deallocate() reclaims that memory.
FixedSize means it only has to be able to handle one value for the first parameter to allocate(), and returns a pointer that be used to construct() a single element
VariableSize means it must be able to handle any value for the first parameter to allocate(), up to the value returned by max_size(), and returns a pointer to the first element of a memory sequence suitable to construct() N elements.
Contiguous means "can be iterated by a C style pointer."
NonDeterministicReclaim, *Hybrid* and *ExpandInPlace* are included as examples of how these concepts can be extended. They are elaborated on in a later section.

AllocatorAdapter

This is the interface that enables an AllocatorStorage and AllocatorManager to be used with a container. When the current standard says "the allocator" it is referring to the AllocatorAdapter. The AllocatorAdapter can only refer to one AllocatorManager.
Only the AllocatorAdapter can be "stateful" or "stateless."

It adapts the functions and typedefs of the AllocatorStorage and AllocatorManager for use in containers.

Table 5 Programmability of AllocatorAdapter

	Stateless	CopySemantic	MoveSemantic
DefaultConstructible	Yes	NullSemantic*	Yes
Swappable	Yes	Yes	No
MoveConstructible	Yes	Optional	Yes
MoveAssignable	Yes	Optional	Yes
CopyConstructible	Yes	Yes	No
CopyAssignable	Yes	Yes	No
EqualityComparable	Yes	Yes	Yes

*NullSemantic means that the default constructor places the object in a null state.

Examples

std::allocator (stateless)
`__gnu_cxx::malloc_allocator (stateless)`
A container allocator that maintains a pointer to an AllocatorManager (CopySemantic)
A container allocator that embeds an AllocatorManager (MoveSemantic, swappable)
A container allocator that embeds an AllocatorStorage (MoveSemantic, not swappable)

Discussion

In practice, the vast majority of AllocatorAdapters are stateless or CopySemantic. In some cases, it is desirable to code the AllocatorManager and even the AllocatorStorage into the AllocatorAdapter. In these cases, the AllocatorAdapter has the minimum programmability available in either AllocatorManager or AllocatorStorage.
MoveSemantic is discussed in a later section.

Current State of Affairs

Intent of the Current Standard

First, we take note of what the standard already says:

Table 40 --Note 224 : *It is intended that `a.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’’.*

We can see that the allocators are preferred to be FixedSize, i.e. for use in node based containers. We will keep this bias moving forward –indeed, it is the canonical example of a allocator matched to a type’s behavior. Indeed, perhaps the most significant performance enhancements obtainable are with supplying node based containers with allocators that support free lists of size T. Furthermore, these same allocators can be easily used inside class specific operator new and delete. They are ubiquitous and easy to code. Even in the stateful allocators, the “state” is often to make these same FixedSize allocators either object specific, thread specific, practical in concurrent data structures, etc. It is likewise easy and practical to make a stateful class specific operator new /delete that can use these same allocators.
However, virtually none of the trade press mentions this intent. Rather, the focus is on navigating the boilerplate, musings about FAR pointers, the curious weird rebind mechanism, and such. Few users are actually providing their node based containers with free list allocators. So at least one implementation just maintains the free list anyway.
To add to the confusion, not all containers can use FixedSize Allocators.

Recommendation

- The node based containers are NOT defaulted to std::allocator, but rather to a free list allocator.

- For each container, explicitly state what the allocator requirements are

Table 6 Standard Containers that take Allocator Argument

	Allocator Manager Requirement	Allocator Manager Options	Allocator Adapter Requirement	Allocator Adapter Options	Notes
vector	VariableSize	ExpandInPlace, Trivial	CopySemantic	MoveSemantic	Contiguous property only applies when Contiguous allocator is used
sring	Contiguous	ExpandInPlace, Trivial	CopySemantic	MoveSemantic	Support for construct/ destroy prohibited
deque	VariableSize	ExpandInPlace, Hybrid	CopySemantic		
list	FixedSize		CopySemantic	MoveSemantic	
forward_list N2231	FixedSize		CopySemantic	MoveSemantic	
set	FixedSize		CopySemantic	MoveSemantic	
multiset	FixedSize		CopySemantic	MoveSemantic	
map	FixedSize		CopySemantic	MoveSemantic	
multimap	FixedSize		CopySemantic	MoveSemantic	
unordered_set	VariableSize	ExpandInPlace, Hybrid	CopySemantic		
unordered_map	VariableSize	ExpandInPlace, Hybrid	CopySemantic		
unordered_multiset	VariableSize	ExpandInPlace, Hybrid	CopySemantic		
unordered_multimap	VariableSize	ExpandInPlace, Hybrid	CopySemantic		

Stateful AllocatorAdapter Programmability

Multithreaded programming has made stateful allocators more than just a curiosity. Let's see what the issues are, and how to fix them. (Almost all of these issues are mentioned in EASTL as well)

DefaultConstructiblity

A problem with stateful AllocatorAdapter is specifying a reasonable value to use for default construction. In general, there is no reasonable default value. This is true of many resource managers. As an illustration, consider the havoc if shared_ptr<T> was defaulted to the address of some global static T, rather than 0. It would always make the pointer dereferenceable – thereby eliminating accidental seg faults. However, calling the default T is likely NOT what you had in mind. Now, root out just where you forgot to initialize the shared_ptr….

The current practice is to disable the default ctor. Why? Consider this:

```
MyAllocatorStorage  mybuf(ptr_beg,ptr_end);//some memory
MyAllocatorManager my_am(mybuf);//OK
```

```
struct MyAllocatorAdapter{
  MyAllocatorAdapter(MyAllocatorManager &);
private:
  MyAllocatorAdapter();//default is meaningless
};
typedef std::container<T, MyAllocatorAdapter> my_t;
typedef std::container<my_t> mc_t;
mc_t tmp;//OK
//tmp.resize(1);//BOOM no compilation
tmp.resize(1, my_t(my_am));//Works for most implementations
```

A novice may think “Well, I could just make MyAllocatorAdapter use malloc as a default.” And indeed it would compile and link. However, note what we just did: malloc may (and probably does) implement completely different policies than MyAllocatorStorage and MyAllocatorManager. And these concepts are essential for program correctness –else you would have just used malloc.

CopyConstruction and CopyAssignable

The DefaultConstructible problem is exacerbated by the fact that the only way to specify AllocatorAdapters is by copy construction. The standard reads:
23.1/8 Copy constructors for all container types defined in this clause copy an allocator argument from their respective first parameters. All other constructors for these container types take an `Allocator&` argument (20.1.5), an allocator whose value type is the same as the container’s value type. A copy of this argument is used for any memory allocation performed, by these constructors and by all member functions, during the lifetime of each container object. In all container types defined in this clause, the member `get_allocator()` returns a copy of the Allocator object used to construct the container

In short, once constructed with an allocator, that allocator does not change. This leads to this curious problem: upon a mutation that increases the size of the controlled sequence, some implementations will default construct an element first, and then move the values in later using a proprietary method similar to a move. So in container<container2> just how are you supposed to get your container2 element’s special allocator copied into the containers controlled sequence? And forget about using any mutating algorithms. My experience is by killing the allocators default ctor, and then much trial and error with the different implementations. (not trivial when coding something like map<special_string,special_set_of_special_vector>,…,specialallocator>)
The perfect solution would be to go through the specification of every mutating member function, and specify whether or not it is allowed to default construct an element. Then a programmer only has to dodge the minefield only once. I think this is forcing the issue a little too much – there are good reasons for each implementation to do what it does. Besides, then you would have to note just which algorithms and adapters (e.g. back_inserter) use which mutators. A good approach is rather: CopyAssign the AllocatorAdapter via operator=, the problem does not completely go away; however, it brings it into the realm of sanity. An application writer can make the AllocatorAdapter default just indicate a runtime error, and then “assign it out” before use. This would mimic the behavior of a container of shared_ptrs -- a shared_ptr can be default constructed to Null, and then assigned before use.

This is safer than the method used by EASTL, which has a “set_allocator” function. With set_allocator, one has to worry about the cases where it is called when the container already has elements.
Furthermore, copying the allocator adapter brings this semantic inline with any other container value, like controlled sequence elements, comparators, hashers, etc. And, allows the algorithms to be used in the normal way, without having to make “stateful allocators” a special case.

Recommendation

- Require that no implementation may call allocate() in the container’s default constructor
- Require that for every container, operator= copy the AllocatorAdapter as well (except if the allocator compares equal)
- Require that for every container, swap exchanges the AllocatorAdapter as well. This allows operator= to be defined in terms of swap, and vice versa.

EqualityComparable

The semantic that “operator== return true iff memory allocator by one can be deallocated by the other” is underspecified. There are several cases where this is true, but otherwise one would say they are unequal:
1. mmap operates on a given file handle, but munmap does not need this filehandle. So any allocator with “file handle state” can deallocate from any other, but they should not compare equal
2. many allocators just take a buffer, and deallocate is a no-op. One allocator takes a buffer to shared memory, the other to the current stack. They could deallocate each other, since they don’t “deallocate” at all. Here again, allocators are not equal

The point here is that instead of trying to reason about the “meaning” of operator==, rather reason about what a container does if in fact they do compare equal. For a consideration of allocators that can reallocate, see this discussion.

Recommendation

- When allocators compare equal, and implementation is free to deallocate allocations from the other.
- When allocators compare equal, an implementation is not required to swap nor assign allocators.
- list::splice has a precondition that allocators compare equal
- EqualityCompare is transitive see Section 3 of N2346

Stateful AllocatorAdapters and Rebind

Now that we have figured out in general just how to get our container to actually use the allocator we specify, now comes the problem: just how do we specify the allocator?

23.1/8 All other constructors for these container types take an `Allocator&` argument (20.1.5), an allocator whose **value type** is the same as the container's **value type**.

This is great, except for the case of stateful FixedSize allocators. Consider

```
MyAllocatorStorage  mybuf(ptr_beg,ptr_end);//some memory
FixedSizeAllocatorManager my_am(mybuf);//supposed to allocate nodes of
T
typedef RegularAllocatorAdapter< FixedSizeAllocatorManager >
                        MyAllocatorAdapter;
typedef std::set<T, MyAllocatorAdapter> my_t;
my_t my_cont(MyAllocatorAdapter(my_am));//OK
```

…but…

How does FixedSizeAllocatorManager know just what size my_cont wants to allocate????

Without breaking encapsulation, there is no way to portably know at compile time. We just wait for the first call to allocate to find out. We need a better way.

This was also noted in EASTL.

Recommendation

- No AllocatorAdapter definition can depend on the names of the type it allocates (i.e. for AllocatorAdapter<T>, AllocatorAdapter cannot depend on a name in T)
- Every container has some identifier making it possible to determine the alignment and size of the allocations required (The EASTL method does not account for alternative pointers, however, but the idea is the same)

(note this presents a problem with containers that use more that one type of allocator (deque , unordered_*) I regret that I have not worked out a acceptable solution for this)

MoveSemantic AllocatorAdapters

One part of the confusion is that the literature confuses the portion of the allocator that actually does the allocating with the portion that adapt this for use in a container. The former is a typical resource manager object, and is naturally move semantic. The latter is almost always (in the stateful case) a copy of a pointer to this resource manager. It is ONLY the case that the adapter and the resource manager are both the same object that this situation occurs. Containers in general cannot tolerate this case – for example, it is hard to implement a unordered_* or a deque that only used one copy of an allocator – unless it forced this allocator not to care about the type of object it allocates. And this situation is antithetical to alternative allocators (and performance minded applications).

One problem with MoveSemantic adapters is how to swap them. Even if the user never calls swap, (either directly or indirectly) some libraries use swap internally, to implement `operator=(), clear(), reserve()` and such.

Another problem with MoveSemantic allocators is that they cannot be easily used in containers that use more than one copy of the allocator, such as deque and unordered_*. Likewise, containers of containers using MoveSemantic allocators, that has the same problems that `auto_ptr` would have i.e. you can't reliably use it with the algorithms. Adding to the headaches, `get_allocator()` can no longer return a copy of the "allocator." `get_allocator()`is used in at least one implementation internally, and also custom adapters that rely on the "copy behavior."

I have found that MoveSemantic adapters make some syntax easier, but rarely does it translate to performance gains or ease of use overall.

Recommendation

- Support for MoveSemantic AllocatorAdapters is implementation dependent

Polymorphic Allocators

This means that at least the functions allocate() and deallocate() are declared virtual. Given that standard container can inquire at compile time about specific types supported by an allocator, it has a distinct advantage over containers that use only polymorphic allocators. Either polymorphic allocators have to be applied carefully to a particular type --thereby negating the polymorphism. Or, a polymorphic allocator has to be able to allocate ANY type --thereby negating any performance tuning that can be applied to a particular types behavior. On top of that, polymorphic allocators cannot be used with relocatable container, since virtually every compiler implements virtuals using pointers. It is a perhaps a coming of age for any IPC programmer to discover that they can't use virtuals reliably in shared memory.

To add to the restrictions, polymorphic allocators cannot tolerate alternative pointers nor alternative accessors.

Interestingly, other that the original HP reference implementation, all implementation correctly handle polymorphic allocators. *The issue is that they are hard to apply, tune and debug*. Yes, it is easier to *compile and link* a polymorphic allocator, but that is where the advantage ends. Even Section 4 of N2387 recommends quite a battery of tests to ensure that the users polymorphic allocators were applied correctly.

Nevertheless, in many cases polymorphic allocators can be advantageous. In these cases, we aren't looking for how to make containers go fast or how to easily debug mistakes – polymorphic allocators can never contend with their templatized cousins in this regard. What we are looking for is the ability for a library to advertise "installable allocators." There is no reason that the specification of std::allocator cannot be pressed into this service, so that a library that advertises this ability can also easily be implemented in terms of standard containers.[7] This does not mean that any part of the standard allocator is specified as virtual (there by negating the space concerns), just that virtuals could easily be used.

To this end, the signature to allocate is modified to deprecate the "const void*" hint (which has very limited applicability) and changed to take alignment and flags parameters. In most typed allocators the alignment parameter (just like the size parameter) can be ignored, but when the allocator specification is pressed for service for use outside a single container, this parameter can be used effectively.

The flags parameter is implementation defined.

See EASTL.

[7] This does NOT imply that containers with polymorphic allocators make good types to use in the interface of such libraries. Almost always, such libraries have abstract base classes as interfaces. Standard containers are still templates, with is problematic at best to mix with abstract base classes, plus one has to worry about the lifetime, origination and behavior of the allocators passed back and forth via the interface containers

Recommendation

- Overload the allocate function to pointer allocate(std::size_t, std::size_t align, unsigned flags);
- The allocate and deallocate functions are non-const

Alternative Pointers

Alternative Pointers has not found much use, mostly because the weasel wording neuters this feature, and each implementation has very different notions of what is possible. (For example, one implementation applies stateful construct and destroy not only to the controlled sequence, but to the pointers themselves.)
There are at least two classes of pointers used: first is those that a container uses for it own internals (i..e node pointers, iterator implementation), and the other is for the signature of construct/destroy/address/data, but otherwise not used by implementations to *store the value* of the pointer.
This code snippet sums it up

```
template<class T, template<class Y> class SmartPtr>
struct custom{
    typedef T* raw_pointer;//could be FAR, etc
    typedef T const* raw_const_pointer;
    typedef T& reference;
    typedef T const& const_reference;
    typedef SmartPtr<T> pointer;
    typedef SmartPtr<T const> const_pointer;
  //  static_assert(is_same<raw_pointer,typeof(&*SmartPtr<T>)>::value)
  //  static_assert(is_convertible<pointer,custom<void>::pointer)
  //  pointer u;u==pointer() means "NULL"
};
```

`raw_pointer` is what is used for the arguments to construct, and destroy. It is traditionally where a user you specify FAR, NEAR, etc. In other words, `raw_pointer` is something that works with extern C. `pointer` is what is used for implementing iterators, in the interface of a container, and pointers that are used as member (e.g. node pointers). This is where a pointer –like class is used.
We note this relationship – *every container that requires a VariableSize AllocatorManager also requires a pointer that is a model of RandomAccessIterator. Otherwise, the pointer could be a model of TrivialIterato*r – in other words, no arithmetic is ever applied to the pointer objects themselves in this case. This mirrors the case with operators new / new[].

shared_ptr

shared_ptr is a model of TrivialIterator, but it will not work in any standard container that could take a FixedSize AllocatorManager (but see forward_list N2231 for a counterexample). auto_ptr and unique_ptr will of course not work at all with the current model.
Why might I want a shared_ptr? It would be possible in theory to make a custom container where iterators are always dereferencable. So allocators themselves should allow it, just not the existing standard containers. (This is similar to the auto_ptr problem – it is possible to make a container that uses auto_ptrs, just that the standard containers

can't use them reliably. And part of the goal is a specification that custom container writers can use as well).
Making a standard container use a shared_ptr as the pointer type is a can of worms – First, one would have to deal with the cycles inherent in the node based containers; second, one would have to figure out a way that the shared_ptr allocator "matched" the containers allocator; third, a deleter would have to be used that told the shared_ptr not only which allocator to delete from, but what the second parameter is to `deallocate`. Fourth, the deleter would have to call `destroy` for you. Fifth, even though the iterator may be correctly dereferenced, they remain invalid otherwise. Now put all this together such that the container works with both T* and shared_ptr<T>. Possible? Yes. Practical? I don't think so.

basic_string

alternative pointers can cause some grief when used with basic_string. This is because char_traits requires "raw pointer" types as function arguments. Therefore, it is unlikely that a fancy pointer (which is typically NOT convertible to its associated raw pointer type) can be passed directly to it. So when an implementation passes a iterator or the internal buffer to a char_traits function, that implementation should always call address() or &* first. This relieves the user of having to define an appropriate char_traits just so fancy pointer could be used.

data()

The return type of data() should be the raw pointer type. Why? Because this function was intended such that it is compatible with extern C functions. In the odd case that a non-contiguous allocator was used (i.e. for vector) then this function is undefined.

Recommendation

- Add language assuring that pointers have null semantics
- Add language assuring that each AllocatorAdapter has the same "template typedefs" for each specialization See N2082 and "The Standard C++ Library"
- with FixedSize AllocatorManagers, it is desired that pointer types that are not models of RandomAccessIterator be allowed for container that could use them. This make programming alternative pointers far easier
- Add language that stated that any pointer type used for list, map, multimap, multiset, and set allow cyclic references
- Separate out the pointer typedefs used for construct/destroy/address from the others. This reduces the boilerplate required
- Add diagnostics to disallow shared_ptr or unique_ptr for `pointer` inside any standard container.
- Require implementations to convert fancy pointer to raw pointer when calling char_traits functions
- Require that constructors and assign operators of alternative pointers are no-fail (this makes it easier to code for alternative accessors)

Alternative Accessors

The standard says, a==b, iff b.`deallocate`(a.`allocate`(N,0),N) is valid. However, nothing can be said about b.`destroy`(a.`construct`(P,V));
N2257 declares that there is no known use for this. Or course the standard discourages ANY use by neutering these functions (as noted in LWG 580). And while many implementations call these functions, few do so in a usable way –vector typically does, but then again writing a variable sized object allocator for vector is difficult.[8] So of course there are few people using these, since taking the trouble is not portable nor well understood.

There are at least two uses cases for construct/destroy:
Relocatable containers: It is desirable to make containers relocatable. To do this we need to store offsets rather than raw pointers.
And indeed offset_ptr fits the bill..

```
double val;
offset_ptr<double> offp(&val);
&*offp==&val;//OK
```

However, I can't do this

```
offset_ptr<double> offp2(a.allocate(1));
new ((void*) offp2) T(val);
```

The remedy is simple – define construct() to handle my special pointer type. However, if the above changes proposed for pointers are adopted, then this point is moot.
Object Pools: I can easily make a case for an allocator that recycles live objects, rather than just raw memory. A suitable construct/destroy would readily handle this, if certain limitation are set on just what a container can do with construct/destroy.
Construct/destroy should only be applied to the elements of the controlled sequence (that is, the "T"). Applying it to the nodes does little good – a user does not want to construct/destroy nodes for the container, and shouldn't. Applying construct/destroy to fancy pointers is also not needed. A fancy pointer supplier is going to have suitable regular constructor / destructors anyway, and is unlikely to then override that behavior with construct/destroy. This rule takes away the issue of constructing and destroying things like comparators, the pointers themselves, temporaries, and every other component of a container
To avoid confusion with stateful allocate/deallocate, make construct/destroy are static functions. This makes issues of "swap" "copy" and "equality" moot, and brings a little sanity. Any state that is needed could be put into the memory referred to by pointer argument.
Construct/destroy use raw pointers as arguments, and not fancy pointer types.
Construct/destroy should not be used in basic_string. basic_string was designed for use with POD types only. Applying construct/destroy to each member is a big performance hit. No implementation actually uses construct/ destroy in basic_string, but this is not mentioned in the standard.[9]

[8] It is possible to wrap up a std::deque for this. See Exercise 10.10 in the "The Standard C++ Library"
[9] For vector of POD types, sometime big performance gains can be realized by using basic_string in lieu of vector – especially in those cases that COW makes sense

When the constructors and assign operators of the alternative pointers are no-fail, this makes code using construct/destroy far easier to implement.

address()? If the language for pointers is adopted, then this is replaced by &*ptr. This simplifies what a container author must be able to support (see section 8 of N2387)

Recommendation

- destroy, construct and address are static members only
- destroy, construct and address are **only** applied to members of the controlled sequence.
- destroy, construct, data and address signature only use “raw” pointers. These can still be parameterized (i.e. to specify FAR, etc) but the typedefs are distinct
- construct/destroy is prohibited from use in basic_string

Example of list code snippet that uses construct/destroy in the manner above. NOTE— the ctor/dtor of ListNode is never called. Here, the ListNode relies on “structural” typing only – the traditional “C++ type expresses behavior” is neutered, delegating this to other functions[10], namely construct/destroy:

```
template<class T, class Alloc>
struct ListNode{
 typedef typename Alloc::template
            rebind<ListNode>::other::pointer node_pointer_type;
 T m_val;
 //m_val is coincident with the start of ListNode -- important!!
 node_pointer_type m_next;
 node_pointer_type m_prev;
private:
//ctor/dtor should not be called
//this conflicts with construct/destroy
      ListNode();//=delete
      ~ListNode();//=delete
};
template<class Alloc>
struct allocholder{
 typedef Alloc value_allocator_type;
 typedef typename Alloc::template
      rebind<ListNode>::other node_allocator_type;
protected:
 node_allocator_type m_nodeallocator;
 allocholder(Alloc const&a):m_nodeallocator(a){}
};
template<class T, class Alloc>
struct list:allocholder{//typical EBCO
      typedef Alloc value_allocator_type;
      typedef typename ListNode<T,Alloc>::node_pointer_type
            node_pointer_type;
//internal use class
```

[10] This is the approach taken in “The Standard C++ Library” but only applied to the T, and not everything else

```
friend struct CreateNode;

//internal member function to create node
node_pointer_type get_newnode(node_pointer_type prev,
                              node_pointer_type next,T const&val) {
 CreateNode newnode(prev,next,val,*this);
 node_pointer_type ret();
 using std::swap;
 swap(newnode.m_newval,ret);
 return ret;
}

//CreateNode does the work of constructing a ListNode
struct CreateNode{
CreateNode(node_pointer_type prev, node_pointer_type next,
           T const&val,list &a):m_parent(a){
 m_newval = a->m_nodeallocator.allocate(1);
 //construct/destroy NOT applied to container internals
 new (static_cast<void*>(&(m_newval.m_next))
               node_pointer_type(next);
 new (static_cast<void*>(&(m_newval.m_prev))
             node_pointer_type(prev);
 //construct/destroy applied to element

 value_allocator_type::construct(
           value_allocator_type::address(m_newval->m_val), val);
}
~CreateNode(){
      //NULL pointer check
 if(m_newval==node_pointer_type())return;
 //call ListNode dtors
 (m_newval.m_prev).~node_pointer_type();
 (m_newval.m_next).~node_pointer_type();
 //deallocate
 a->m_nodeallocator.deallocate(
      node_allocator_type::address(m_newval),1);
}
   node_pointer_type m_newval;
   m_parent &m_alloc;
}
//other node creation/deletion functions follow the same intent
};//list
```

max_size

We note that AllocatorStorage has two concepts, bounded and unbounded. “unbounded” simply means “up to the limits imposed by the system.” The max_size() has little meaning in the unbounded case. In the bounded case, it means “how many objects can I put into the container?”

Recommendation

- for every container that maintains the strong guarantee (i.e. list), for every mutating member that can insert more than one element at a time into the controlled sequence, test max_size first.

- Each container max_size function must subtract from max_size the number of allocations it uses for its own internals (i.e. list, tree base nodes)
- max_size is const when not static

Support / Requirement Levels

Not all containers need to support all features. However, instead of ambiguous "weasel wording" this is a language by which a container author can advertise their support. While it should be mandatory that the standard containers all portably support a minimum of features, specialty container authors are can describe the features they support, graded on level of implementation difficulty.
SupportLevel0 – no allocator required (tr1::array)
SupportLevel1 – custom allocator changes not possible, or on a global basis (ICU, bitarray, locales)
SupportLevel2 – static allocator adapter, no alternative pointers, no alternative accessors (the current standard)
SupportLevel3 – SupportLevel2, plus stateful allocator adapter (most custom implementations, and most shared_ptr's)
SupportLevel4 - SupportLevel3, plus alternative pointers
SupportLevel5 - SupportLevel4, plus static alternative accessors (applied to value only)
SupportLevel6 - SupportLevel5, plus stateful alternative accessors (applied to pointers and value)

(users that need support beyond this are likely to implement custom data structures)

SupportOptionA – ExpandInPlace Allocators
SupportOptionB – Trivial Allocators
SupportOptionC – Hybrid Allocators

Also, some specialty containers have additional requirements on allocators, beyond FixedSize, VariableSize, or Contiguous

RequirementA – Concurrent / NonDeterministicReclaim (BDW)
RequirementB – Serialized (gcc mtalloc)
RequirementC -- Unbounded
RequirementD -- Relocatable

Recommendation

- all standard variable size containers support a minimum SupportLevel5. This covers all known use cases.
- string and shared_ptr uses SupportLevel3

The Code

This sample allocator implementation does not require any interface changes to existing containers, nor any code changes (other than meeting minimum support levels).

There is still an “allocator” template, but users should not have to replace this in any conceivable use case. It is for backward compatibility only.
There are three new templates: A default “AllocatorStorage” class, for defining fancy pointers and construct/destroy, one static “Manager Class,” and a “Stateful Adapter” class, for easily specifying stateful pools. The replacement allocator class is modified to adapt the new templates.
The idea is to make the manager classes easy to replace - -this is the majority of the alternative allocator usage. Only when a user changes the pointer type or use object pools does a lot of boilerplate come into play, and even then less than the current model.
No examples of alternative pointers or alternative accessors are provided in this paper, in the interest of saving space – the required pointer definitions and allocators alone would more that double the length of the paper (but the allocator boilerplate is a small fraction of it!!)

AllocatorStorage

This is the place that the pointer typedef’s and construct/destroy live

```
template<class Y>
struct AllocatorStorage {
      template<class T> struct rebind{
            typedef AllocatorStorage<T> other;
      };
      typedef Y value_type;
      typedef value_type* raw_pointer;
      typedef value_type const* raw_const_pointer;
      typedef value_type& reference;
      typedef value_type const& const_reference;
      typedef std::size_t size_type;
      typedef std::ptrdiff_t difference_type;
      typedef value_type* pointer;
      typedef value_type const* const_pointer;
      static void construct(raw_pointer ptr, value_type const& val) {
            new(static_cast<void*>(ptr))value_type(val);
      }
      static void destroy(raw_pointer ptr1) {
            ptr1->~value_type();
      }
};
template<>
struct AllocatorStorage<void> {
      template<class T> struct rebind{
            typedef AllocatorStorage<T> other;
      };
      typedef void value_type;
      typedef value_type* raw_pointer;
      typedef value_type const* raw_const_pointer;
      typedef value_type* pointer;
      typedef value_type const* const_pointer;
      typedef std::size_t size_type;
      typedef std::ptrdiff_t difference_type;

};
//requirements
```

```
//pointer u,raw_pointer v
// is_same<raw_pointer,typeof(&*u)>::value==true
// AllocatorStorage<T>::pointer a; AllocatorStorage<void>::pointer b;
// b(a);//is valid
// pointer n();
// n==pointer();//means "null pointer"
// pointer is model of TrivialIterator, or
// better depending on container
// pointer must have CopySemantics
// pointer must allow cyclic references
```

AllocatorManager

This is the place that the pool managers live. Its default looks like:

```
struct AllocatorManagerBase {
      typedef  AllocatorStorage<void>::size_type size_type;
      typedef  AllocatorStorage<void>::raw_pointer void_pointer;
      static void_pointer allocate(size_type req,
            size_type align=size_type(-1),unsigned flags=0) {
            return ::operator new(req);}
      static void deallocate(void_pointer d,size_type ) {
            ::operator delete(d);}
      static bool equal_to(AllocatorManagerBase const&)  {
            return true;
      }
      static size_type max_size(){
            return size_type(-1);
      }
};
template<class T>
struct AllocatorManager:AllocatorManagerBase{
      template<class U>struct rebind{
            typedef AllocatorManager<U> other;
      };
      template<class U> AllocatorManager(AllocatorManager<U> const&){}
      AllocatorManager(){}
};// max_size always returns number of bytes available
```

The reason for the separation is to demonstrate to users that this can easily be done without templates. The “rebind” mechanism is still required however in the general case.

AllocatorAdapter

There are two adapter templates. One is to make stateful AllocatorManagers easy to specify. The other is for backward compatibly with existing containers:

```
template<class T, class Pool, class Storage=AllocatorStorage<void> >
struct StatefulAdapter{
      typedef typename Storage::size_type size_type;
      typedef typename Storage::difference_type difference_type;
      typedef typename Storage::template
              rebind<Pool>::other::pointer
                   impl_pointer_type;
      typedef typename Storage::template
                   rebind<void>::other::raw_pointer void_pointer;
      template<class U>struct rebind{
            typedef StatefulAdapter<U,Pool,Storage> other;
      };
      void_pointer allocate(size_type req,std::size_t align,
            unsigned flags) {
            return m_Manager->allocate(req,align,flags);
      }
      void_pointer allocate(size_type req,
            unsigned flags=0) {
            return m_Manager->allocate(req,flags);
      }

      void deallocate(void_pointer d,size_type n ) {
            m_Manager->deallocate(d,n);
      }
      bool equal_to(StatefulAdapter const&_r) const {
            return m_Manager==_r.m_Manager;
      }
      size_type max_size()const {
            return m_Manager->max_size();
      }
      StatefulAdapter(Pool const& b)
            :m_Manager(const_cast< Pool *>(&b)){}
      StatefulAdapter():m_Manager(Pool::get_instance()){}
      StatefulAdapter(StatefulAdapter const  & b)
        :m_Manager(b.m_Manager){}
      StatefulAdapter& operator=(StatefulAdapter const& r){
            m_Manager=r.m_Manager;
            return *this;
      }
      template <class U>
      StatefulAdapter (StatefulAdapter <U, Pool,
             Storage >
                  const& r)
      : m_Manager (r.m_Manager) {
      }

      void swap(StatefulAdapter&r){
            using std::swap;
            swap(m_Manager,r.m_Manager);
      }
private:
      template<class X,class Y,class Z> friend struct StatefulAdapter;
      impl_pointer_type  m_Manager;
};
template<class X,class Y,class Z>
void swap(StatefulAdapter<X,Y,Z>&l,StatefulAdapter<X,Y,Z>&r){
  l.swap(r);
}

template <class T,class Manager=AllocatorManager<T>,
           class Storage=AllocatorStorage<T> >
struct AllocatorAdapter:Manager {
    typedef Manager base_type;
      typedef T value_type;
      typedef Storage storage_policy;
      typedef typename storage_policy::pointer pointer;
      typedef typename storage_policy::const_pointer const_pointer;
      typedef typename storage_policy::raw_pointer raw_pointer;
      typedef typename storage_policy::raw_const_pointer
                      raw_const_pointer;
      typedef typename storage_policy::reference reference;
      typedef typename storage_policy::const_reference const_reference;
      typedef typename storage_policy::size_type  size_type;
      typedef typename storage_policy::difference_type difference_type;
      static raw_pointer address(reference _val)
      {
            return (&_val);
      }
      static raw_const_pointer address(const_reference _val)
      {
            return (&_val);
      }
#if defined(NEWSTYLE)
      static void construct(raw_pointer ptr, const_reference val)
      {
            storage_policy::construct(ptr, val);
      }

      static void destroy(raw_pointer ptr1) {
            storage_policy::destroy(ptr1);
      }
#else
      static void construct(pointer ptr, const_reference val)
      {
            storage_policy::construct(address(*ptr), val);
      }

      static void destroy(pointer ptr1) {
            storage_policy::destroy(address(*ptr1));
      }
#endif
    pointer allocate(size_type n)  {
            using boost::alignment_of;
            return pointer(static_cast<raw_pointer>(
                  this->base_type::allocate( n*sizeof (T)
                        ,alignment_of<T>::value )) );
      }
      void deallocate (pointer x,size_type n)  {
            this->base_type::deallocate(address(*x),n);
      }

      size_type max_size()const {
            return this->base_type::max_size()/sizeof(T);
      }
      bool operator==(AllocatorAdapter const &r)const {
            return this->base_type::equal_to(r);
      }
      bool operator!=(AllocatorAdapter const &r)const {
            return !((*this) ==r);
      }
      template<class OtherType>
      AllocatorAdapter(OtherType const & _buf):base_type(_buf) {
      }
      AllocatorAdapter(AllocatorAdapter const&r ):base_type(r){}
    AllocatorAdapter(Manager const&r ):base_type(r){}

      template<class U>struct rebind {
            typedef AllocatorAdapter<U,
                        typename base_type::template
                                          rebind<U>::other,
                        typename storage_policy::template
                                          rebind<U>::other> other;
      };
      template <class U>
      AllocatorAdapter(AllocatorAdapter<U,Manager,Storage>
                  const& r)
      : base_type(r) {
      }
      template<class U>
      AllocatorAdapter<T,Manager,Storage>&
      operator=(AllocatorAdapter<U,Manager,Storage>const&r)
      {
            this->base_type::operator=(r);
            return (*this);
      }
      void swap(AllocatorAdapter &r){
        using std::swap;
        swap(static_cast<base_type&>(*this),
            static_cast<base_type&>(r));
      }
      AllocatorAdapter() {}

};
template <class Manager,class Storage =AllocatorStorage<void> >
struct AllocatorAdapter<void,Manager,Storage> {
      typedef void value_type;
      typedef Storage storage_policy;
      typedef typename storage_policy::pointer pointer;
      typedef typename storage_policy::const_pointer const_pointer;
      typedef typename storage_policy::raw_pointer raw_pointer;
      typedef typename storage_policy::const_raw_pointer
const_raw_pointer;
      typedef typename storage_policy::reference reference;
      typedef typename storage_policy::const_reference const_reference;
      typedef typename storage_policy::size_type  size_type;
      typedef typename storage_policy::difference_type difference_type;
      template<class U>struct rebind {
            typedef AllocatorAdapter<U,
                  typename Manager::template rebind<U>::other,
            typename storage_policy::template rebind<U>::other > other;
      };
};
template <class T,class U,class Manager,class Storage>
bool operator==(AllocatorAdapter<T,Manager,Storage>
            const&l,AllocatorAdapter<U,Manager,Storage> const&r) {
      return l.operator==(r);
}
template <class T,class U,class Manager,class Storage>
bool operator!=(AllocatorAdapter<T,Manager,Storage>
            const&l,AllocatorAdapter<U,Manager,Storage> const&r) {
```

```
	return !(l==r);
}
```

This adapter makes it easy to specify "polymorphic allocators" if desired, merely by changing the Manager class.

Examples

These are a few examples. The idea is that the most common cases –like just changing what allocate and deallocate do in the static case – requires the least boilerplate. Even in the more difficult cases, the boilerplate is still far less than that required by the current design.
Template Typedefs and constructor forwarding would make this easier still.

Malloc Allocator

An all time classic

```
template<class T>
struct Malloc_Alloc{
	template<class U>struct rebind{
		typedef Malloc_Alloc<U> other;
	};
	template<class U> Malloc_Alloc(Malloc_Alloc<U> const&){}
	Malloc_Alloc(){}
	typedef  std::size_t size_type;
	static void* allocate(size_type req,
		size_type align=size_type(-1),unsigned flags=0) {
		return malloc(req);}
	static void deallocate(void*p ,std::size_t ) {
		free(p);
		}
	static bool equal_to(Malloc_Alloc const&)  {
		return true;
	}
	static size_type max_size(){
		return size_type(-1);
	}
};
template<class T>struct MallocAlloc{
	typedef AllocatorAdapter<T,Malloc_Alloc<T>  > type;
};
```

Usage is like this:

```
std::vector<int,MallocAlloc<int>::type >  myvec;
```

Polymorphic Allocator

When alternative accessor nor pointers are needed, this is a useful way to write libraries that can use "installable" memory managers determined at runtime. Almost always, these allocators are used to allocate heterogeneous types unknown to the allocator author, so in general they are hard to tune and debug. But for particular applications, they can be advantageous, given the alternative of redefining global operator new and rebuilding everything.

This example also uses a shared_ptr to manage the lifetime of the allocator instance. It looks like this:

```
//the abstract base class
//get_instance() returns a default
struct Poly_Alloc {
      typedef std::size_t size_type;
      virtual void* allocate(size_type req,
            size_type align=size_type(-1),unsigned flags=0)=0;
      virtual void deallocate(void*p ,size_type )=0;
      virtual size_type max_size()const=0;
      static shared_ptr<Poly_Alloc> get_instance();
};
// a suitable default
struct DefaultPoly_Alloc: Poly_Alloc{
      void* allocate(size_type req,
            size_type =size_type(-1),unsigned =0) {
            return ::operator new(req);}
      void deallocate(void* d,size_type ) {
            ::operator delete(d);
      }
      size_type max_size()const {
            return size_type(-1);
      }
};
//get_instance, just a regular singleton
void no_op(void*){}
shared_ptr<Poly_Alloc> Poly_Alloc::get_instance() {
      static DefaultPoly_Alloc defalloc;
      static shared_ptr<Poly_Alloc> defalloc_ptr(&defalloc,no_op);
      return defalloc_ptr;
}
//something to tell the StatefulAdapter to use a shared_ptr
//this does NOT change the definition of allocator::pointer
template<class Y>
struct SharedPtrSpec : AllocatorStorage<Y> {
      template<class Z> struct rebind{
            typedef SharedPtrSpec<Z> other;
      };
      typedef shared_ptr<Y> pointer;
};
//put it all together
template<class T>struct PolyAlloc {
 typedef AllocatorAdapter<T,StatefulAdapter<T,Poly_Alloc,
                              SharedPtrSpec<void> > > type;
};
```

Usage is

```
typedef std::set<int,std::less<int>,PolyAlloc<int>::type > myset_t;
```

Stack Memory Allocator

An example of a stack memory allocator, that can be shared amongst other containers, or is unique to one (i.e. could be MoveConstructible):

```
//StackAllocatorBase is the machinery
template<class T, std::size_t N> struct StackAllocatorBase {
      static const std::size_t my_align =alignment_of<T>::value;
      void* allocate(std::size_t req, std::size_t /*align*/
                    =std::size_t(-1),
               unsigned /*flags*/=0) {
          std::size_t const areq=req%my_align ?
                         req+my_align-req%my_align : req;
          if (unsigned((reinterpret_cast<char*>(&m_buf)
                                   +sizeof(m_buf))-m_ptr)<=areq)
               throw std::bad_alloc();
          void* ret=m_ptr;
          m_ptr+=areq;
          return ret;
     }
     void deallocate(void*, std::size_t) {
     }
     std::size_t max_size() const {
          return sizeof(m_buf);
     }
protected:
     StackAllocatorBase()
     : m_ptr((char*)&m_buf) {}
     //wierdness needed to get this to work in the "unique" case
     StackAllocatorBase(StackAllocatorBase const&)
          :m_ptr((char*)&m_buf) {}//just a default ctor
     void operator=(StackAllocatorBase const&) {/*do nothing*/}
private:
     aligned_storage<sizeof(T)*N,alignment_of<T>::value> m_buf;
     char * m_ptr;
};
```
Now, for a version that can be shared and works all containers:
```
//StackAllocShareable will be used with the StatefulAdapter
template<class T, std::size_t N>
struct StackAllocShareable :StackAllocatorBase<T,N> {
     static StackAllocShareable*get_instance() {
          return 0;
     }
     StackAllocShareable() {
     }
private:
#if !defined(__GNUC__)
     //gcc wants to see the copy ctor
     //but it should not be required
     StackAllocShareable(StackAllocShareable const&);//=delete
#endif
     void operator=(StackAllocShareable const&);//=delete
};
```
And one that is unique, that works with many containers[11]
```
//StackAllocUnique will be used with the AllocatorAdapter
template<class T, std::size_t N>
struct StackAllocUnique :StackAllocatorBase<T,N> {
     template<class U> struct rebind {
          typedef StackAllocUnique<U,N> other;
     };
     static bool equal_to(StackAllocUnique const&) {
          return false;
     }
```

[11] Theoretically, any container that makes only one copy of the allocator argument

```
};
```

And a way to put this together:

```
template<class T, std::size_t N> struct StackAlloc {
      typedef StackAllocShareable<T,N> stype;
      typedef AllocatorAdapter<T,StatefulAdapter<T,stype > > ctype;
      typedef AllocatorAdapter<T,StackAllocUnique<T,N> > mtype;
};

typedef StackAlloc<int,1000> myalloc_t;
```

Now, the user does this for the “unique” case:

```
typedef std::vector<int,myalloc_t::mtype > mvec_t;
mvec_t mvec;
mvec.resize(3,1);
mvec_t mvec2;
mvec2=mvec;
```

And for the shared case, one does this:

```
//this vector assigns the allocator in operator=
typedef vector<int,myalloc_t::ctype > cvec_t;
cvec_t cvec((myalloc_t::ctype(myalloc_t::stype())));
cvec.resize(3,1);
cvec_t cvec2;
cvec2=cvec;
cvec_t cvec3(cvec2.begin(),cvec2.end(),cvec2.get_allocator());
```

The “shared” version works with every container in every implementation commonly in use (provided that you don’t call a default as I did here), and the “unique” version works with many containers on many implementation, but has more “gotcha’s” -- like O(N) swap for instance.

Shared Memory Container

Lets modify the “stack” example above to make it use a deque, and usable in shared memory. To do this, we cannot rely on the value of any pointer. We can only use offsets. With offset_ptr, we have what we want.

```
template<class Y>
struct RelocatableStorage {
      template<class U>struct rebind{
         typedef reloc_node_policy <U> other;
       };
      typedef Y value_type;
      typedef offset_ptr<Y> pointer;
      typedef offset_ptr<Y const> const_pointer;
      typedef value_type*                 raw_pointer;
      typedef value_type const*     raw_const_pointer;
      typedef value_type& reference;
      typedef value_type const& const_reference;
      typedef std::size_t size_type;
```

```
        typedef std::ptrdiff_t difference_type;
        static void construct(Y* ptr,  Y const&  val)
        {
              new(static_cast<void*>(ptr))Y(val);
        }
        static void destroy(Y* ptr1) {
              ptr1->~Y();
        }
};
template<>
struct RelocatableStorage<void> {
        template<class T> struct rebind{
              typedef reloc_node_policy<T> other;
        };
        typedef void value_type;
        typedef value_type* raw_pointer;
        typedef value_type const* raw_const_pointer;
        typedef offset_ptr<value_type>pointer;
        typedef offset_ptr<value_type const> const_pointer;
        typedef std::size_t size_type;
        typedef std::ptrdiff_t difference_type;
};
```

Change StackAllocatorBase to use max alignment, and change m_ptr to read,

```
    RelocatableStorage<char>::pointer  m_ptr;
```

Now, with the proviso that std::deque is going to actually use the pointer specified by the allocator, and of course T is relocatable as well:

```
template<class T, std::size_t N>
struct RelocAlloc {
        typedef StackAllocShareable<T,N> stype;
        typedef AllocatorAdapter<T,
                    StatefulAdapter<T,stype,RelocatableStorage<T> >,
                    RelocatableStorage<T> > allocator_type;
};
template<class T,std::size_t N>
struct RelocDeque
:RelocAlloc<T,N>::stype
,std::deque<T,typename RelocAlloc<T,N>::allocator_type >{
        typedef std::deque<T,typename RelocAlloc<T,N>::allocator_type >
                    base_type;
        RelocDeque():base_type(typename
              base_type::allocator_type(*this)){};
};
```

This is a deque that be properly placed in shared memory, without “fixing” the addresses.

```
char* sharedmem=mmap(0,sizeof(RelocDeque<int,200>),/*args*/);
RelocDeque<int,200>& relocdeq=*new(sharedmem)RelocDeque<int,200>;
```

Object Pool

With a suitable allocator manager (not shown here), one does this:

```
template<class Y>
struct ObjectStorage :AllocatorStorage<Y> {
        //two user defined functions to "hook"
```

```
      //into the specialty allocator manager
      template<class U>
      static void alloc_destroy(U* ) {
      //do nothing
      }
      static void* alloc_construct(void*ptr) {
            return ptr;
      }
};
template<>
struct ObjectStorage<void>
      :AllocatorStorage<void> {};
template<>
struct ObjectStorage<std::string>
       :AllocatorStorage<std::string> {
      //this is now how a container controls
      //an elements lifetime
      static void construct(pointer ptr,
               value_type const& val) {
            *ptr=val;
      }
      static void destroy(value_type* ptr1) {
            ptr1->clear();
      }
      static void alloc_destroy(pointer ptr) {
            ptr->~value_type();
      }
      static value_type* alloc_construct(void*ptr) {
            return new(static_cast<void*>(ptr))value_type;
      }
};
```

For example, during allocate, if there are no objects on the free list, an allocator manager will allocate a new node, and the call `alloc_construct`. At some time after the container is destroyed, then `alloc_destroy` is called. The typedefs to get this into a list are more involved, but amounts to this :

```
typedef ObjectStorage<std::string> ObjectStringStorage;
//ListNode is made public, so that Alignment and Size can be inquired
typedef ListNode<std::string, ObjectStringStorage >
           ObjectStringListNode;
typedef ObjectAllocator<std:string,sizeof(ObjectListNode)
            ,alignment_of(ObjectListNode), ObjectStringStorage>
          ObjectStringNodeManager;
typedef AllocatorAdapter<std::string,ObjectStringNodeManager,
            ObjectStringStorage > ObjectStringNodeAdapter;
list<std::string,ObjectStringNodeAdapter>
          objectstringlist;//at last!!!!
```

`ObjectAllocator` does the work, and the code is not show here –however anybody reading this should be able to make an example. This requires that the nodes that list manages can be inquired as to size and alignment, and that the list follows the guidelines for construct/destroy set forth earlier.

Implementation Defined

Now that we have cleaned up some of the specification for the typical cases of alternative allocators, and saw what we could do only if we found a suitable implementation, we can move on to other interesting use cases.

Trivial Allocators

Many implementations can accommodate Trivial Allocators for vector and string, as long as you don't expect to

1. swap the container
2. ever cause a reallocation after the initial one

Imagine I had a AllocatorAdapter that combined the AllocatorManager and AllocatorStorage together:

```
template<class T,std::size_t N>
struct Trivial{
    aligned_storage<sizeof(T)*N,alignmentof<T>::value> buf;
    void* allocate(std::size_t ){return &buf;}
    void deallocate(void*,std::size_t){}
    static std::size_t max_size(){return sizeof(buf);}
    bool operator==( Trivial const&)const{return false;}
};
vector<double,AllocatorAdapter<double,
              Trivial<double,100> > > myvec (100);
//do anything with myvec as long as capacity() never exceeds 100
```

Almost as fast as std::array<double,100> except its variable size!!
Furthermore, with a proper selection of pointer an implementation can easily make myvec relocatable.
There is some work needed to make this portable and practical, however.

Concurrent Containers

Although not part of the standard, it is not hard to imagine that in the near future a concurrent container library would be proposed. ("concurrent" meaning concurrent reads and writes)
Concepts of thread access models

	Refinement of	
Sequential	None	Multiple threads must be serialized
Concurrent	Sequential	Does not include calls to acquire more blocks of memory from system

Of course, every standard container requires the Sequential model. The foundation of the STL is the Iterator concept. Data structures that have iterators cannot be concurrent -- Why? Because to meaningfully "iterate" through anything requires an arbitrary amount of steps to complete before modifications are allowed. And this is not "concurrent." However, many data structures can be concurrent, such as stacks and queues. These are far easier to implement (but still difficult) when the allocator is likewise concurrent and reclaim is deferred to some arbitrary point in the future. This is the motivation for NonDeterministicReclaim.

Details are beyond the scope of this paper, but I have in my toolbox containers that use concurrent allocators.

ExpandInPlace

The idea here is that containers that use VariableSize allocators can attempt to allocate more memory, while keeping the existing memory valid, much like realloc().
In this case, you would prefer that the AllocatorAdapter is not copied during operator=. This isn't a problem in the stateless case.
In the stateful case, it is still preferable to copy (or move) the allocator, since a container can never be sure of just what part of the state should be copied and what should not. As an example, consider an allocator implementation for use on Linux – it's state is comprised of the file descriptor, protection flags, sync state, lock state. The allocator performs mmap, munmap, mremap, msync and mlockall. Methods like msync and mlockall are performed via extensions.
Such an allocator can ExpandInPlace, and is clearly stateful. Does the "ExpandInPlace" semantics negate all the other semantics of this allocator when operator= is invoked? A rule is – if the allocators compare equal, then the implementation is free to ExpandInPlace using the current allocator. ExpandInPlace:;realloc should not be a member of allocator, but rather an implementation that support it should implement a traits class, to inquire if an allocator support this capability. The default of course is to reallocate then deallocate, and an implementation should correctly handle this case.

Hybrid Allocators

For certain values of allocate() this allocator performs like a FixedSize allocator, otherwise it performs like VariableSize. This is useful for deque and unordered_*. The trouble is knowing just how many special values exist, and what those values are. Additionally, and implementation could take advantage of ExpandInPlace for the VariableSize allocations.
Due to the varying ways that unordered_* and deque can be implemented, this is hard to standardize. One solution is to use a non-zero hint parameter to signal the allocator when a FixedSize allocation is requested.

Scoped Allocator Semantics

This is perhaps the logical conclusion of polymorphic allocators discussed earlier. The idea is that when a container has an element that can take an allocator argument, the containers allocator is passed to the element, and the element-to-be uses that allocator to construct a copy of itself. In the case that the object is NOT part of a container, then the allocator is defaulted, or user supplied. The application should not care on just what allocator is selected.
In this method, any container must use any allocator. One can easily see this greatly restricts what performance enhancements we can do. The chosen allocator must be the least common denominator (unbounded and contiguous, equivalent to the default), and all containers act accordingly. Furthermore, the container doesn't actually have to use the allocator you specified. Indeed, it is difficult to write such an general purpose allocator[12].

[12] See http://www.cs.umass.edu/~emery/heaplayers.html for a library that indeed accomplishes this feat

But this style does have one interesting advantage: In the case where it is important that "containers of containers" all use the same stateful AllocatorStorage, this makes it easy to code, as long as you disable the default ctor. To this end an adapter can be used that simplifies the otherwise tedious syntax. For example

```
template<class T,class S=vector<T,PolymorphicAllocator> >
struct scopedsequenceadapter:protected S{
//all non-mutators fwd to S
void resize(size_t _sz, T const& _Val){
// alloc_traits  determines if the T takes an allocator argument
// detail::choose_ctor uses the correct constructor
     if (size() < _sz)
   this->S::insert(end(), _sz - size(),
       detail::choose_ctor<value_type,allocator_type,
            alloc_traits<value_type>::value >(_Val,get_allocator()));
   else{
      this->S::resize(_sz);
   }
};
```

This method also changes the complexity guarantees of swap and splice, and makes it very hard to code the normal cases where you *do* care about the performance and other properties of the AllocatorManager.

Bibliography

Most references are included as hyperlinks in the text. Virtually all of this paper is the result of studying source code, due to the dearth of commentary on C++ allocators. These are a few libraries and papers that stood out:

Apache Software Foundation stdcxx
http://incubator.apache.org/stdcxx/
This is RogueWave's Open Source donation.
This open source library that has excellent support for alternative allocators

Plauger, et al "The C++ Standard Template Library"
http://www.amazon.com/exec/obidos/ASIN/0134376331
Simply the best book for people wanting to implement their own high quality containers. Contains full source code (alas not open source) and commentary of a container library that has excellent support for alternative allocators. Working through the "very hard" exercises alone qualifies anyone as a C++ black belt.

The Dinkum Allocator Library
http://www.dinkumware.com/manuals/?manual=compleat&page=allocators.html
A great reference implementation for allocator authors.

Heaplayers
http://www.cs.umass.edu/~emery/heaplayers.html
The best studies in applying general purpose allocators to whole applications. Library code included. This is similar to the types of applications that might use polymorphic allocators.

Veldhuizen, Todd "Active Libraries and Universal Languages"
http://www.cs.chalmers.se/~tveldhui/papers/2004/dissertation.pdf
While not specifically relate to allocators, Dr. Veldhuizen makes the argument that libraries take an active role in their own optimization. This is the same observation that Dr Stepanov had that inspired the port of STL to C++.

About the Author

I work with High Frequency Trading Systems at UBS. I have been programming C++ applications for over 12 years, ranging from embedded systems to large SMP machines. For the past five years, I have been studying just what that "allocator" argument to standard containers is good for.