Status of allocators in the present WP

Matthew Austern (`austern@sgi.com`)

November 13, 1996

1 Introduction

The September '96 working paper still has many unresolved issues related to allocators. Resolving these issues one at a time might be possible, but it will certainly require many small fixes and at least a few large fixes. This paper simply lists some of the problems, so that people can get an idea of how much work would be involved in fixing them. I do not claim to have identified every important issue.

Many of the problems relate to the fact that the generic allocator requirements are seriously underspecified. This is not a trivial matter: generic programming is only possible if the requirements for a generic concept are specific enough so that clients (in this case, containers) can be written using no assumptions other than the ones in the requirements table. Nor is it an easy matter to fix: even for a relatively simple generic concept, such as Forward Iterator, it takes a great deal of thought to write down requirements that are stringent enough so that they provide all necessary operations, but weak enough so that they do not rule out useful implementations. Many apparently simple issues in lists of generic requirements have very far-reaching ramifications, some of which are not immediately obvious.

Some of these issues, but not all, were previously listed in my reflector message `c++std-lib-4879`.

2 Problems related to nested typedefs

Allocators define the nested types `pointer`, `const_pointer`, `reference`, and `const_reference`. The semantics of these types are seriously underspecified, and not completely consistent.

2.1 Reference types

- It is impossible to define a user-defined type that has the same semantics as `T&`: in particular, C++ does not provide any way to overload "`operator.`", the member selection operator. The only possible definition for `Allocator<T>::reference` within the C++ type system is `T&`.

- Generalized references will not work as arguments to function templates whose arguments are `T&` or `const T&`: even if there is a conversion from `Allocator<T>::reference` to `T&`, the conversion will not be used for the purposes of template type deduction. And, of course, a declaration of the form

  ```
  template<class Allocator, class T> void f(Allocator<T>::reference);
  ```

  is not legal C++.

- The requirements table doesn't say whether or not `reference` is guaranteed to be a POD type.

- The requirements table doesn't say whether or not `Allocator<T>::reference` is convertible to `Allocator<T>::const_reference`.

- The requirements table doesn't provide any way of converting an `Allocator<T>::const_reference` to an `Allocator<T>::reference`. That is, it provides no equivalent of `const_cast`.

- If class `D` is derived from class `B`, then the requirements table doesn't say whether or not `Allocator<D>::reference` is convertible to `Allocator<B>::reference`. Additionally, it does not provide any mechanism for conversions in the other direction. That is, it provides no equivalents of derived-to-base conversion, `static_cast`, `dynamic_cast`, or `reinterpret_cast`.

- The requirements table says that `Allocator<T>::reference` is convertible to `T&`, but doesn't say whether or not `T&` is convertible to `Allocator<T>::reference`.

- If `r` is of type `Allocator<T>::reference`, the requirements table doesnt't say whether or not `&r` is a valid expression. Assuming that it is a valid expression, the requirement table doesn't say what its type is. (Reasonable options include `T*` and `Allocator<T>::pointer`.)

2.2 Pointer types

2.2.1 Semantics of `pointer` and `const_pointer`

- If `p` is of type `Allocator<T>::pointer` or `Allocator<T>::const_pointer` and `n` is of type `Allocator<T>::difference_type`, then the requirements table doesn't say whether or not the expressions `p++`, `++p`, `p--`, `--p`, `p + n`, `p - n`, `n + p`, `n - p`, `*p`, `p[n]`, and `n[p]` are valid expressions. If they are valid, it does not define their return type or their semantics.

- The requirements table implies, but does not actually say, that `p1 - p2` is a valid expression whose return type is `Allocator<T>::difference_type`. In any case, it does not define the semantics or preconditions of this operation.

- The requirements table doesn't say whether or not `Allocator<T>::size_type` and `Allocator<T>::difference_type` are guaranteed to be the same for every specialization of `Allocator`, nor does it say whether or not `sizeof(Allocator<T1>::pointer) == sizeof(Allocator<T2>::pointer)`.

- The requirements table doesn't guarantee that `operator->` is defined for `Allocator<T>::pointer` if `T` is a class type.

- Pointers to member are entirely missing from the requirements table.

- If `p1` and `p2` are of type `Allocator<T>::pointer`, the requirements table doesn't say whether or not the expressions `p1 == p2`, `p1 != p2`, `p1 < p2`, `p1 > p2`, `p1 <= p2`, and `p1 >= p2` are valid exprssions. If they are valid, it does not define their semantics. (The issue about semantics is highly nontrivial: consider the case where `p1` and `p2` are returned by different invocations of `A.allocate()`, or where one is returned by `A1.allocate()` and the other by `A2.allocate()`. It's not obvious what the behavior of `operator==` ought to be.)

- Ordinary pointers can be used as iterators; in particular, `iterator_traits<T*>` is guaranteed to be defined. However, there is no such guarantee for `iterator_traits<Allocator<T>::pointer>`.

- The requirements table doesn't say whether or not `Allocator<T>::pointer` is guaranteed to be a POD type. This has implications for unions, static versus dynamic initialization, and other issues.

- The requirements table doesn't say whether or not `Allocator<T>::pointer` is convertible to `Allocator<T>::const_pointer`.

- The requirements table doesn't provide any way of converting an `Allocator<T>::const_pointer` to an `Allocator<T>::pointer`. That is, it provides no equivalent of `const_cast`.

- If class `D` is derived from class `B`, then the requirements table doesn't say whether or not `Allocator<D>::pointer` is convertible to `Allocator<B>::pointer`. Additionally, it does not provide any mechanism for conversions in the other direction. That is, it provides no equivalents of derived-to-base conversion, `static_cast`, `dynamic_cast`, or `reinterpret_cast`.

- The requirements table doesn't say whether or not values of type `Allocator<T>::pointer` and `Allocator<T>::const_pointer` satisfy the invariant `p == &*p`. (It's not even possible to tell whether `p` and `&*p` necessarily have the same types.)

2.2.2 Conversions from `pointer` to `T*`

The requirement that `Allocator<T>::pointer` be convertible to `T*` and `void*` raises many unanswered questions. It also prohibits many potentially useful types of allocators. Note that eliminating this requirement, however, would require changes to several parts of the WP: it is used by `Allocator<T>::operator new`, `Allocator<T>::construct`, `Allocator<T>::destroy`, the specialized algorithms in §20.4.4, `basic_string` (Clause 21), and possibly other library components.

- The requirements table says that `Allocator<T>::pointer` is convertible to `T*`, but doesn't say whether or not `T*` is convertible to `Allocator<T>::pointer`.

- If `p` is of type `Allocator<T>::pointer`, the requirements table doesn't say whether or not `static_cast<void*>(static_cast<T*>(p))` is the same as `static_cast<void*>(p)`, or whether `static_cast<T*>(static_cast<void*>(p))` is the same as `static_cast<T*>(p)`.

- The requirements table doesn't say how the conversion interacts with pointer arithmetic. For example, if `p` is of type `Allocator<T>::pointer` and `n` is of type `Allocator<T>::difference_type`, the requirements table doesn't say whether or not `static_cast<T*>(p) + n` is guaranteed to be the same as `static_cast<T*>(p + n)`. Note that this issue has far-reaching implications: an affirmative answer to it means that converting a single pointer from `Allocator<T>::pointer` to `T*` automatically results in the conversion of every pointer in the array that the pointer belongs to. This would be a very significant implementation constraint. A negative answer, however, would break `basic_string`.

- The requirements table doesn't say whether or not the mapping is one-to-one. That is, if `p1` and `p2` are of type `Allocator<T>::pointer` and `p1 != p2`, it doesn't say whether it's guaranteed that `static_cast<T*>(p1) != static_cast<T*>(p2)`;

- The requirements table doesn't say whether or not pointers belonging to different allocator types must convert to unequal pointers. That is: if `p1` is of type `Allocator1<T1>::pointer` and `p2` is of type `Allocator2<T2>::pointer`, it doesn't say whether or not it is guaranteed that `static_cast<void*>(p1) != static_cast<void*>(p2)`. (This is an extraordinarily strong guarantee. Nevertheless, some allocator clients seem to implicitly rely on it.)

- The requirements table doesn't say whether or not `static_cast<T*>(p) == static_cast<T*>(p)`. That is, if you convert the same `Allocator<T>::pointer` twice, possibly separated by intervening operations, it doesn't say whether or not the two conversions yield the same `T*` value.

- The requirements table doesn't say what the lifetime is of the pointer returned by `static_cast<T*>(p)`. (The end of the enclosing full-expression? The lifetime of `p`? The local scope?)

3 Interaction with other parts of the library

- An object allocated using `a.allocate()` must be deallocated using `a.deallocate()`. The `auto_ptr` template, however, has no provision for an allocator instance. This means that `auto_ptr` can't be used for pointers allocated using allocators, which, in turn, means that `auto_ptr` can't be used for implementing containers.

- The container adaptors `stack`, `queue`, and `priority_queue` have constructors that take an Allocator argument. The container requirements, however (Table 75) do not provide a way to construct a container from an allocator. (Possible fix: change the container adaptor constructors so that they take a container instead of an allocator; all containers must have copy constructors.)

- The `basic_string` template is parameterized by an allocator, so its underlying memory is pointed to by some `Allocator<charT>::pointer`. However, all of `basic_string`'s primitive operations use `char_traits`, and `char_traits`'s members take arguments of type `charT*` rather than `Allocator<charT>::pointer`. (This is necessarily the case, since the allocator and traits classes are two separate and unrelated template parameters.) This means that `basic_string` must rely very heavily on the conversion from `Allocator<charT>::pointer` to `charT*`. Unless that conversion obeys some very strong constraints, `basic_string` is broken.

- There is no mechanism for `get_temporary_buffer` to use allocators. (Footnote 200 (on page 20-18) says that there is such a mechanism, but that footnote doesn't make sense. It's left over from an earlier allocator design.)

- The `valarray` template makes no provision for allocators. It would be easy to parameterize them with respect to an allocator class, but making them deal with allocator instances would be more ambitious.

4 Allocator instances

The fundamental problem is that it's very hard to specify the semantics of containers while taking the possibility of multiple allocator instances into account, and it's hard to write containers that are robust under this possibility. This is an especially serious problem in the case of user-defined containers.

- The allocator requirements table doesn't explicitly say that an allocator is guaranteed to have a default constructor. However, note that `vector<int, my_alloc>` doesn't satisfy the container requirements unless `my_alloc` has a default constructor.

- Allocating a container with allocator placement `new` yields counterintuitive results, since the container components aren't allocated with the argument that was passed to placement `new`, but with either the default allocator or the allocator associated with the initializer. There are no good workarounds for this problem, which is especially serious in the case of `pair`. How, for example, do you use a shared memory allocator to allocate a pair consisting of a 10 MB fixed size array and a character?

- Constructing nested containers (*e.g.* `list<vector<int, my_alloc>, my_alloc>`) is difficult. The default constructor for `vector<int, my_alloc>` uses the default allocator instance `my_alloc`, rather than the instance associated with the containing `list`. The only workaround is to create a "prototype" instance of the nested container (e.g. `vector`) first, and to avoid any operations, such as certain versions of `insert`, that use `vector`'s default constructor. This is clumsy and may be expensive. It will ensure that most code will not be robust for multiple allocator instances, because it's much easier to write code that isn't.

- The WP doesn't explicitly say that `basic_string::append` uses the left-hand-side allocator. Assuming that this is the intended requirement, however, it prohibits any implementation that performs lazy concatenation. This implementation constraint effectively requires inefficient code. Similar considerations apply to `basic_string::replace`, and to other string operations.

- The WP defines an assignment operator as part of the allocator requirements, but it is insufficiently explicit about the semantics of assignment. In particular, it doesn't say what happens to memory that was allocated using the allocator on the left hand side of the assignment. This is especially significant in the case of allocators that are associated with system resources, such as allocators that implement persistence or shared memory.

- The WP doesn't specify whether or not memory allocated using a particular allocator instance can outlive that instance's duration. For example, is this code valid?

```
{
  my_alloc<int>::pointer p;
  {
     my_alloc<int> a;
     p = a.allocate(1);
     *p = 3;
  }
  cout << *p << endl;
}
```

- The WP doesn't say whether `basic_string::operator= (const basic_string& S)` copies `S`'s allocator, or whether it leaves the left-hand operand's allocator unchanged.

- In general, the WP is silent on the question of whether a container's allocator is fixed at compile time or whether there are any circumstances under which it is permitted to change. (The former seems to be the intention, but it is nowhere stated.)

- Avoiding the extra space overhead to store an allocator in every container requires a tricky hack; it's uncertain whether, with today's compiler technology, this hack will work in real code. This is an especially serious problem if, as has been suggested on the reflector, each `list` node is required to carry its own allocator instance.

- Many container operations must be performed differently depending on whether or not two allocator instances compare equal; this requires more complicated member functions. In the case of alocators whose instances always compare equal, it isn't clear whether this extra code is actually removed by the compiler.

- There is no reasonable way to specify the allocator used by the binary string concatenation `operator+`. In principle it ought to behave like a constructor and take an allocator parameter, but it can't, because it's a binary operator. The WP isn't explicit about which allocator is used, but it implies that `x + "a"` and `"a" + x` are allocated using different allocators. This problem generalizes to binary operations on user-defined containers.

- The WP is silent about how `list::splice` and `list::merge` work in the case of unequal allocators. Reflector discussion shows that there is no consensus about the appropriate semantics.

- To allocate a complete data structure through an allocator instance, it will often be necessary to allocate pieces of it with an `operator new` that uses an allocator instance. This requires placement new. But placement `operator new[]` is essentially broken in the current WP. There is no portable way to deallocate an array allocated using placement `operator new[]`.

- Consider sorting the vector `V`, where `V` is declared as `vector<vector<int, alloc1>, alloc2>`. Each of the vectors contained within `V` may have a distinct allocator instance. What is the semantics of `sort(V.begin(), V.end())`? Note that specifying the semantics of `swap` and `operator=` is insufficient, since it is unspecified which of those primitives, if either, is used by `sort`. Not also that this issue is not unique to `sort`: it also applies to `rotate`, `reverse`, `stable_partition`, and many other algorithms.