Doc No: N3165=10-0155
Date: 2010-10-15
Authors: Pablo Halpern
Intel Corp..
phalpern@halpernwightsoftware.com

Allocator Requirements: Alternatives to US88

Contents

National Body comments and issues

This paper addresses comment US 88 to the July, 2010 FCD.

Document Conventions

All section names and numbers are relative to the, August 2010 WP, **N3126**.

> Existing working paper text is indented and shown in dark blue. Edits to the working paper are shown with ~~red strikeouts for deleted text~~ and green underlining for inserted text within the indented blue original text.

Comments and rationale mixed in with the proposed wording appears as shaded text.

Requests for LWG opinions and guidance appear with light (yellow) shading. It is expected that changes resulting from such guidance will be minor and will not delay acceptance of this proposal in the same meeting at which it is presented.

Background

In comment US 88, it is correctly pointed out that the simplification to the allocator interface made possible by the use of `allocator_traits` in the FCD weakens the Allocator

Requirements relative to the C++03 standard. The consequence of this weakening is that an allocator that is written to the minimal FCD specification may not work with a container that is written to the C++03 specification. The author of US 88 proposes that the Allocator requirements be restored to the C++03 specification and goes on to describe how an allocator author is allowed to (and is perhaps encouraged to) inherit the boilerplate for an allocator from std::allocator.

No existing code is broken by the weakening of the Allocator requirements, but it is philosophically problematic, in general, to either strengthen or weaken named requirements between revisions of the standard. Strengthening a named requirement can cause a type that conforms to the requirement before the change to no longer conform after the change, and thus no longer work with components that depend on the named requirement. Weakening a named requirement can cause a component that depends on the requirement before the change to no longer work with parameters that conform to the (weaker) requirement after the change (as in the case of the Allocator requirement and containers that depend on Allocator). Thus weakening a named requirement, even if existing code does not break, is problematic because future code could be incompatible with existing code.

The question before the committee is whether this philosophical breakage is serious enough in the case of Allocator to warrant making a change to the WP. If so, what should that change be?

In this paper, I argue that US 88 should be resolved as NAD. However, if the committee decides that the philosophical breakage is unacceptable, I also present an alternative resolution that keeps the Allocator requirements minimal and also provides a model for changing other named requirements in the future.

Argument for NAD

Size of the problem

As previously stated, no code written to the C++03 standard will break because of the changes in the Allocator requirements. The standard containers defined in the FCD are, of course, written to the weaker specification, and would therefore continue to work with existing user-defined allocators written to the stronger C++03 specification. A new user-defined allocator, written to the FCD specification, however, may not work with an existing user-defined container written to the C++03 specification of Allocator.

Viewed as a grid, the allocator-container compatibility chart looks as follows:

	user-defined C++03 container	user-defined FCD container	standard FCD container
user-defined C++03 allocator	yes	yes	yes
user-defined FCD allocator	**no**	yes	yes
standard FCD allocator	yes	yes	yes

In my experience, most user-defined containers either don't use allocators or are built on top of standard containers and, therefore, inherit their support of allocators. The inability to create portable stateful allocators in the past has resulted in allocators being largely ignored by authors of user-defined containers. I assert that the single case that causes incompatibility between a user-defined, FCD-compliant allocator and a user-defined C++03-compliant container is extremely rare.

Problem with the resolution proposed in US 88

The reduced set of requirements for allocators dramatically reduces the complexity of simple user-defined allocators. To retain some of this simplicity, the current proposed resolution suggests that users inherit from `std::allocator`. Applying this suggestion, a simple allocator might look like this:

```
template <class Tp>
class MyAllocator : public std::allocator<Tp>
{
public:
  template <class U>
  struct rebind
  {
    typedef MyAllocator<U> other;
  }

  MyAllocator(ctor args);

  template <class U>
  MyAllocator(
      const MyAllocator<U>&);

  Tp* allocate(std::size_t n);
  Tp* allocate(std::size_t,
               const_pointer);
  void deallocate(Tp*, std::size_t);
};

template <class Tp>
bool operator==(const MyAllocator&, const MyAllocator&);
template <class Tp>
bool operator!=(const MyAllocator&, const MyAllocator&);
```

It is my assertion that inheriting from `std::allocator` was never sound engineering practice and has resulted in subtle bugs, particularly related to incorrectly inheriting `rebind`. Some of these bugs were not seen until a template library was made available to other users. The situation is worse in the case of stateful allocators because a stateful allocator derived from std::allocator is likely to have a bug in `operator==` as well as in `rebind`. Many of the changes to allocators in the FCD, including the elimination of weasel words and the addition of allocator propagation traits, are designed to make stateful allocators more usable. Thus, the use of `std::allocator` as a base class does not offer the same simplification as the reduced allocator requirements and is likely to confuse a novice allocator author and sour him/her on allocators entirely. (I reject any assertion that allocators are generally written by experts only. I have seen a number of otherwise-reasonable attempts at allocators where relative novices were derailed by exactly the subtleties I describe.)

Without the use of any inheritance tricks, we can compare a simple allocator written to the current FCD requirements against a simple allocator written to the C++03 requirements:

FCD requirements

```
template <class Tp>
class MyAllocator
{
public:
  typedef Tp value_type;

  MyAllocator(ctor args);

  template <class U>
  MyAllocator(
      const MyAllocator<U>&);

  Tp* allocate(std::size_t n);
  void deallocate(Tp*, std::size_t);
};

template <class Tp>
bool operator==(const MyAllocator&,
                const MyAllocator&);
template <class Tp>
bool operator!=(const MyAllocator&,
                const MyAllocator&);
```

C++03 requirements

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

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

  MyAllocator(ctor args);

  template <class U>
  MyAllocator(
      const MyAllocator<U>&);

  size_type max_size() const;
  pointer address(Tp&) const;
  const_pointer
      address(const Tp&) const;
```

```
  void construct(pointer, Tp&) const;
  void destroy(pointer) const;

  Tp* allocate(std::size_t n);
  Tp* allocate(std::size_t,
               const_pointer);
  void deallocate(Tp*, std::size_t);
};

template <class Tp>
bool operator==(const MyAllocator&,
                const MyAllocator&);
template <class Tp>
bool operator!=(const MyAllocator&,
                const MyAllocator&);
```

Although the code on the right may not appear horribly onerous, it must be remembered that each of the 12 functions needs an implementation, no matter how trivial. It is a matter of opinion as to whether this extra work constitutes a minor inconvenience or a major inconvenience, but we must also consider how many programmers would be inconvenienced verses how many would benefit. The original proposed resolution would require that all allocator authors put in this extra effort, whereas leaving the FCD alone would require such an effort only by those who wish to keep compatibility with the few user-defined containers that were written strictly to the C++03 allocator requirements.

Alternate Resolution: Legacy Allocator Requirements

Allocator requirements comprise only one out of many named sets of requirements in both the C++03 standard and the FCD. If named requirement sets were to be frozen for all time, it would hamper the Standard Committee's efforts to create a more expressive standard C++ language and library in the future. Yet, carelessly changing requirements will cause significant problems for programmers and will slow the adoption of future standards. What is needed is a graceful way to change requirements going forward. One possible way is to preserve existing old requirements while defining the new requirements as a subset or superset of the old ones. If the new requirements are intended to replace the old ones, then the old ones might be renamed and perhaps deprecated. This approach allows interfaces to older generic code to continue to be specified in terms of the old requirements, at least during the transition. The approach described here can also be applied to named Concepts, if and when concepts are eventually added to the standard.

The proposed wording below applies this deprecation idea to the allocator requirements by reviving the original C++03 Allocator requirements, renaming them to LegacyAllocator requirements, and moving them to Appendix D (deprecated features).

Proposed Wording

D.11 LegacyAllocator Requirements

Insert the following section into appendix D (Compatibility features):

D.11 `LegacyAllocator` Requirements [depr.legacy.alloc]

The library defines a standard set of requirements for *legacy allocators*, which are a superset of the requirements for *allocators* defined in [allocator.requirements], Table 42. Specifically, the following optional allocator members described in Allocator requirements are *required* (i.e., not optional) in a type that conforms to the LegacyAllocator requirements:

- `pointer`
- `const_pointer`
- `size_type`
- `difference_type`
- `rebind`
- `allocate` with hint
- `max_size`

It is likely that `reference` and `const_reference` will be added back to table 42. If so, then they should be added to the list above and removed from the table below.

Table *xyz* defines additional requirements for legacy allocators. Variables used within this table are described in Table 41.

Table *xyz* – LegacyAllocator requirements (in addition to Allocator requirements)

Expression	Return type	Assertion/note pre-/post-condition
`X::reference`	`T&`	
`X::const_reference`	`const T&`	
`a.address(r)`	`X::pointer`	`a.address(r) == p`
`a.address(s)`	`X::const_pointer`	`a.address(s) == q`
`a.construct(p, t)`	(not used)	*Effect:* Constructs an object of type `T` at `p`. (See note A, below)
`a.destroy(p)`	(not used)	*Effect:* Destroys the object at `p`. (See note A, below)

Note A: The actual signatures of the `construct` and `destroy` functions may be different from that specified in the table, provided that they may be called with arguments of the specified types.

The `LegacyAllocator` requirements may be used to describe parameters of user-defined templates that use allocators but not `allocator_traits`. All of the allocators defined in the standard for which `pointer` is the same as `T*` conform to the `LegacyAllocator` requirements (in addition to the `Allocator` requirements). Reference to these requirements is deprecated and they are not referenced elsewhere in this standard. [*Note:* users are encouraged to use `allocator_traits` in order to make their code dependent on only the (weaker) `Allocator` requirements – *end note*]

Other Alternatives Considered, but Rejected

It would be possible to create a `legacy_allocator_adaptor` that would allow a C++0x allocator to be used with a container written according to the C++03 Allocator requirements. A partial implementation of this idea exists and I believe it is viable, but I consider the problem that such a class would solve to be small enough that it is not worth burdening the standard with another class. However, if the committee believes that such a class is the preferred resolution, I would be willing to draft wording and create a reference implementation.

Another way to simplify C++03 compatibility is to provide a base class that supplies the boilerplate for C++03 Allocator requirements. Such a base class has been fully implemented, in fact, and even supports generalized pointer types. This approach was seen as inferior to the adaptor idea, however, because a base-class requires that C++03 compatibility be considered at class-definition time whereas an adaptor can be used to create compatibility after the fact.

Acknowledgements

Thanks to Howard Hinnant and Daniel Krugler for supporting my efforts to keep allocators simple for the allocator author. Thanks to Doug Gregor for helping me understand how Concepts and requirements can evolve between revisions of the standard. Thanks to John Lakos and others for reviewing and editing this paper.

References

N2982: Allocators post Removal of C++ Concepts

N3102: ISO/IEC FCD 14882, C++0X, National Body Comments