Compromise allocator proposal

Matthew Austern (austern@sgi.com) Nathan Myers (ncm@cantrip.org)
Sean Corfield (sean@ocs1td.com)

November 13, 1996

Abstract

This is a compromise proposal for fixing allocators, inspired by N1008 = 96-0190 and by the discussion in the Kona allocator technical session.

1 Clause 20 changes

Replace Table 41 (“Descriptive variable definitions”) with the following table.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>T, U</td>
<td>Any type</td>
</tr>
<tr>
<td>X</td>
<td>An Allocator class for type T</td>
</tr>
<tr>
<td>Y</td>
<td>The corresponding allocator class for type U</td>
</tr>
<tr>
<td>t</td>
<td>A value of type <code>const T&amp;</code></td>
</tr>
<tr>
<td>a, a1, a2</td>
<td>Values of type X</td>
</tr>
<tr>
<td>b</td>
<td>A value of type Y</td>
</tr>
<tr>
<td>p</td>
<td>A value of type X::pointer, obtained by calling a1.allocate, where a1 == a.</td>
</tr>
<tr>
<td>q</td>
<td>A value of type X::const pointer obtained by conversion from a value p.</td>
</tr>
<tr>
<td>r</td>
<td>A value of type T&amp; obtained by the expression *p</td>
</tr>
<tr>
<td>s</td>
<td>A value of type const T&amp; obtained either by the expression *q or by conversion from r.</td>
</tr>
<tr>
<td>u</td>
<td>A value of type Y::const pointer, either obtained by calling Y::allocate or else 0.</td>
</tr>
<tr>
<td>n</td>
<td>A value of type X::size_type.</td>
</tr>
</tbody>
</table>

Change Table 42 (“Allocator requirements”), in clause 20.1.5 [lib.allocator.requirements] as follows.

- Change the specification columns of `pointer` and `const_pointer` to read, respectively, “pointer to T” and “pointer to const T”.
- Change the specification columns of `reference` and `const_reference` to read, respectively, `T&` and `const T&`.
- In the line defining `rebind<>`, change the `return type` column entry to “Y”, and the `note` column entry to:
For all U (including T), Y::rebind<T>::other is X.

- Delete the lines defining operators new, delete, new[], and delete[], and the line defining operator=.
- Change the expression “X a1(a2);” to “X a1(a2);”, and change the corresponding semantics column to read “post: Y(a) == b;”.
- Change the semantics column of operator== to read “Returns true if both storage allocated from each can be deallocated via the other”.
- Add a line describing a default constructor, just before the line that describes the copy constructor. The expression column reads X(). The return type column is empty. The semantics column reads “Creates a default instance”.
- Add a footnote to the description of a.allocate that reads as follows.
  
  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.”

Delete paragraph 3 of §20.1.5 [lib.allocator.requirements].

Add the following two paragraphs to the end of §20.1.5 [lib.allocator.requirements]:

Implementations of containers described in this International Standard are permitted to assume that their Allocator template parameter meets the following two additional requirements beyond those in Table 42.

- All instances of a given allocator type are required to be interchangeable and always compare equal to each other.
- The typedef members pointer, const pointer, size type, and difference type are required to be T*, T const*, size_t, and ptrdiff_t, respectively.

Implementors are encouraged to supply libraries that can accept allocators that encapsulate more general memory models and that support non-equal instances. In such implementations, any requirements imposed on allocators by containers beyond those requirements that appear in Table 42, and the semantics of containers and algorithms when allocator instances compare non-equal, are implementation-defined.

In §20.4 [lib.memory], delete the definitions of operators new, delete, new[], and delete[] from the Header <memory> synopsis.

In §20.4.1 [lib.default.allocator], delete the assignment operator, and operator new. Add a member declaration:

\[
\text{allocator} \text{(const allocator&)} \text{ throw();}
\]

Delete from §20.4.1 [lib.default.allocator] and §20.4.1.2 [lib.allocator.globals] operators new, delete, new[], and delete[].

Delete §20.4.1.3 [lib.allocator.example].

2 Clause 21 changes

- In §21.3.5.2 [lib.string::append] and in §21.3.5.3 [lib.string::assign], remove the Allocator& argument.
• In §21.3.5.8 [lib.string::swap], change the **Complexity** clause (paragraph 3) to read “constant time”.

• In §21.3.6 [lib.string.ops], and in §21.3 [lib.basic.string], change the declaration of member get_allocator() to return “**allocator type**”. Change the description to: “Returns: a copy of the Allocator object used to construct the string.”

3 Clause 23 changes

• In Table 75 (which is in §23.1 [lib.container.requirements]) delete the lines that define the type **allocator type** and the expression a.get_allocator(). In the lines that define the expressions a.swap(), a.size(), and a.max_size(), change the entry in the complexity column to “(Note A)”. Add, after the table, “Those entries marked (Note A) should have constant complexity.” Delete the operational semantics specification for assignment.

• Add to paragraph 8 of §23.1 [lib.container.requirements]:

  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 the declarations of queue, priority_queue, and stack, in, respectively, §23.2.3.1 [lib.queue], §23.2.3.2 [lib.priority_queue], and §23.2.3.3 [lib.stack], remove the get_allocator() member function. Remove the allocator_type member typedef. Add a member typedef: `typedef Container container_type;`. In the constructor replace the Allocator constructor argument with `const Container& = Container();`.

• In §23.2.3 [lib.container.adapters], add the following:

  The container adapters each take a Container template parameter, and each constructor takes a Container reference argument. This container is copied into the Container member of each adapter.

• In §23.2.3.2 [lib.priority_queue], add to the second priority_queue constructor a final argument: “`const Container& = Container();`”. In the default constructor, replace the description of the Effects with

  Effects: `Initializes c with y and comp with x; then calls make_heap(c.begin(), c.end(), comp).`

In the second, template, constructor, change the Effects: paragraph to read:

  Effects: `Initializes c with y and comp with x; then calls c.insert(c.end(), first, last); and finally calls make_heap(c.begin(), c.end(), comp).`