Doc. No: X3J16/97-0040
WG21/N1078
Date: 2 June 1997
Project: Programming
Language C++

Reply to: Matt Austern
austern@sgi.com

ALTERNATE POINTER TYPES

DISCUSSION

This is a proposal to relax the restrictions on pointer types in user-defined allocators. It does not address the issue of non-equal allocator instances.

This discussion is based on X3J16/97-0018R1 = WG21/N0156R1, "Allocators and alternative pointer types, revision 1". Major differences: (1) In the earlier paper I identified alternative resolutions in a number of cases; here, I'm proposing a single solution. (2) I identified several issues related to the lifetime of references to elements pointed to by user-defined "pointers". I now believe that this was largely a red herring. The real issue is object identity: an important guarantee seems to be missing both for C pointers and for iterators. This proposal only addresses the missing iterator guarantee, since the pointer guarantee is a core language issue.

The part of this proposal that I like the least is the mechanism for downcasting, performing static casts (e.g. casts from Allocator<void>::pointer to Allocator<T>::pointer), and casting away constness. I dislike making up new syntax, and introducing new member functions. I don't think we have a choice, though. Casts are essential, and a requirement that casts of Allocator<T>::pointer use the same syntax as ordinary casts would be tantamount to saying that Allocator<T>::pointer has to be T*.

OBJECT IDENTITY

These requirements should go in 24.1.3 [lib.forward.iterators], immediately after Table 75.

-- If a == b then either a and b are both dereferenceable, or else neither is dereferenceable.

-- If a and b are both dereferenceable, then a == b if and only if *a and *b are the same object.

CHANGES IN CLAUSE 20

In Table 31, Descriptive variable definitions, add two new lines.

Variable	Definition
p1	Value of type X::pointer, possibly null.
q1	Value of type X::const_pointer, possibly null.
v1	Value of type Y::pointer, possibly null.
u1	Value of type Y::const_pointer, possibly null.

In Table 32 (Allocator requirements)

Change the description of X::size_type to
A type that can represent the size of the largest object in the allocation model, and that can represent every non-negative value of X::difference_type. X::size_type and Y::size_type are the same types.

Change the description of X::difference_type to
A type that can represent the difference between any two pointers in the allocation model. X::difference_type and Y::difference_type are the same types.

Add to the assertion/note column of the X::pointer description:
a mutable random access iterator whose value type, difference type, pointer type, reference type, and iterator category are, respectively, X::value_type, X::difference_type, X::value_type*, X::reference, and random_access_iterator_tag. X::pointer has an automatic conversion to T* and to X::const_pointer.

Add to the assertion/note column of the X::const_pointer description:
a constant random access iterator whose value type, difference type, pointer type, reference type, and iterator category are, respectively, X::value_type, X::difference_type, const X::value_type*, X::const_reference, and random_access_iterator_tag. sizeof(X::pointer) == sizeof(X::const_pointer). X::const_pointer has an automatic conversion to const T*.

Delete the sentence "The result is a random access iterator" from a.allocate()'s assertion/note column, and add:
Postcondition: if the return value is denoted p, then p + n is a past-the-end iterator and all of the pointers in the range [p, p + n) are dereferenceable iterators. None of the pointers in the range [p, p + n) are null pointers, and p + n may be a null pointer only if n == 0.

Add to the assertion/note column of the a.deallocate() description:
deallocate() may not throw exceptions.

Change "x.construct", in the expression column, to "a.construct".
Change "x.destroy", in the expression column, to "a.destroy".
(This corrects an editorial error. "x" is meaningless, since it is not found anywhere in Table 31.)

Add to the assertion/note column of the a.destroy() description:
destroy() may not throw exceptions.

Remove the member function "address" from Table 32.

Change paragraphs 4 and 5 of section 20.1.5 to read as follows:

Implementations of containers described in this International Standard are permitted to assume that their allocator template parameter meets the following additional requirement beyond those in Table 32.

-- All instances of a given allocator type are required to be interchangeable and always compare equal to each other.

Implementors are encouraged to supply libraries that can accept allocators that support non-equal instances. In such implementations, any requirements imposed on allocators beyond those requirements that appear in Table 32, and the semantics of containers and algorithms when allocator instances compare non-equal, are implementation-defined.

Add the following new entries to Table 32.

Expression	Return type	Assertion/note/ pre/post-condition
static_cast<X::pointer>(x)	X::pointer	x is a constant integral expression that evaluates to 0. The return value is a null pointer. Every null pointer compares equal to every other null pointer of the same type.
static_cast<Y::pointer>(p1)	Y::pointer	Requirement: T* has an automatic conversion to U*. [Note: examples are derived-to-base conversion, and casting to a void pointer.] Postcondition: if the return value is cast back to X::pointer, it will compare equal to p1.
static_cast<Y::const_pointer>(q1)	Y::const_pointer	Requirement: const T* has an automatic conversion to const U*. Postcondition: if the return value is cast back to X::const_pointer, it will compare equal to q1.
X::do_static_cast(v1)	X::pointer	Requirement: static_cast from U* to T* is

valid.

X::do_dynamic_cast(v1) dynamic_cast	X::pointer	Requirement: from U* to T* is valid.
X::do_static_cast(u1) static_cast	X::const_pointer	Requirement: from const U* to const T* is valid.
X::do_dynamic_cast(u1) dynamic_cast	X::const_pointer	Requirement: from const U* to const T* is valid.
X::do_const_cast(p1)	X::const_pointer	Postcondition: p1 == do_const_cast(p1).

Add a note at the end of Table 32:

> For any values of type X::pointer and X::const_pointer, valid pointer operations (i.e. operations described in Table 32 or in Tables 73 through 77, where operands satisfy the applicable preconditions) may not throw exceptions.

Add do_static_cast, do_dynamic_cast, and do_const_cast, as static members, to the default allocator in section 20.4.1. Remove both versions of allocator::address(). Add a throw() specification to allocator::destroy() and allocator::deallocate().

STRUCTURE-PRESERVING CONVERSIONS

Option 1:

Add the following sentence at the end of Table 32.

> The conversion from X::pointer to T*, and from X::const_pointer to const T*, is structure-preserving. That is, static_cast<T*>(p + n) == static_cast<T*>(p) + n.

Option 2:

Add the following text following paragraph 2 in section 21.3 [lib.basic.string]:

> The template parameter Allocator is required to conform to the requirements of an allocator (section 20.1.5), and to satisfy the additional requirement that Allocator::pointer, Allocator::const_pointer, Allocator::size_type, and Allocator::difference_type are, respectively, charT*, const charT*, size_t, and ptrdiff_t.