Doc. No: X3J16/97-0018R1
WG21/N1056R1
Date: 13 March 1997
Project: Programming Language C++
Reply to: Matt Austern
austern@sgi.com

ALLOCATORS AND ALTERNATIVE POINTER TYPES, Revision 1

This discussion of issues related to alternative pointer types has been updated based on discussions at the Nashua meeting. It is largely an attempt to identify and categorize issues; we need to understand the problems before we try to come up with working paper wording to solve them.

(My guess is that what we'll end up with is some minor changes to tables 31 and 32, and one new table and a bit of new text in section 20.1.5.)

I. GENERAL POINTER OPERATIONS

A. Sizes.

These are simple guarantees that ought to be there and that don't seem to be controversial, but that are missing from the current WP text.

Allocator::difference_type is a signed integral type, and Allocator::size_type is an unsigned integral type that can represent every non-negative value of Allocator::difference_type.

For every X, Allocator::rebind<X>::other::size_type and Allocator::rebind<X>::other::difference_type are the same types as Allocator::size_type and Allocator::difference_type.

For every allocator X, sizeof(X::const_pointer) == sizeof(X::pointer).

[An earlier version of this paper suggested the requirement that, for every X, sizeof(Allocator::rebind<X>::other::pointer) == sizeof(pointer). This suggestion is wrong, because it is not always true even for ordinary C pointers.]

B. Pointers as random access iterators

The WP says that pointers are random access iterators, but doesn't give enough detail. Here are some suggested ways to fill in the gaps. Again, nobody on the library working group objected to these proposals.

Allocator::pointer is a mutable random access iterator whose value type, difference type, pointer type, reference type, and iterator category are, respectively, Allocator::value_type, Allocator::difference_type, Allocator::value_type*, Allocator::reference, and random_access_iterator_tag.

Allocator::const_pointer is a constant random access iterator whose value type, difference type, pointer type, reference type, and iterator category are, respectively, Allocator::value_type, Allocator::difference_type, const Allocator::value_type*, Allocator::const_reference, and random_access_iterator_tag.

[Note: Why do I say that Allocator::pointer's pointer type is T*, instead of saying that it's Allocator::pointer itself? It's because a random access iterator's pointer type is really only used in a single place: it is the return type of operator->(). If X is a user-defined type, then it's illegal for the return type of X::operator-> to be X itself. The rules in 13.5.6 make it clear that that would give an infinite regress.]

C. Memory blocks.

What are the preconditions for pointer arithmetic? The WP doesn't say.

An Allocator::pointer is very much like a C pointer, in that an Allocator::pointer always points into some specific memory block. (That is, a block of n elements allocated by Allocator::allocate(n).) My suggested resolution, then, is that we adopt the familiar C rules. It is always valid to compare two pointers for equality or inequality, but other operations are valid only within a single memory block. I won't waste time listing the C rules in detail: we've all used C. The relevant parts of the C standard are sections 6.3.6, 6.3.8, and 6.3.9.

C guarantees that every array has a nondereferenceable past-the-end pointer. It is absolutely crucial that we also have this guarantee for Allocator::pointer, and memory blocks allocated by Allocator::allocate(), as well. Vector<> depends on past-the-end pointers, and probably other components do too.

Two options for how to handle this, which are probably equivalent but which will look very different. The choice between them should probably be determined by which one can be expressed more concisely and clearly.

Option 1: List these C rules explicitly.
Option 2: Say that an Allocator::pointer is a random access iterator on a specific range (one returned by Allocator::allocate()), and that that range is terminated by an off-the-end iterator.

The library working group agreed in principle that some statement of this sort was a good idea.

II. NULL POINTERS

In several places (including vectors and singly-linked lists), null pointers are the only reasonable implementation. In other places (including trees), they are far more convenient than other representations. Null pointers are used by the HP, SGI, Rogue Wave, and Plum Hall implementations. Further, some of the wording in the allocator requirements table explicitly addresses null pointers. Eliminating null pointers is not a sensible option; the issues, then, are how to obtain a null value of Allocator::pointer, and what operations to provide that support null pointer values.

A. Syntax

Three options for how to obtain a null value.

Option 1. For every allocator X, a constant integral expression that evaluates to 0 is convertible to a null value of type X::pointer or X::const_pointer.

Option 2. Every allocator X contains a static member variable X::null of type X::const_pointer. (It has to be const_pointer instead of pointer, of course.)

Option 3. Use the default constructor. So, for example, the syntax for obtaining a null pointer of type Allocator::pointer would be something like "typename Allocator::pointer()".

Arguments in favor of option 1. First, "principle of least astonishment". C and C++ programmers have known for many years that a null pointer is spelled 0. Why break it? Second, every STL implementation that I have looked at (HP, SGI, Rogue Wave, and Plum Hall) assumes that 0 is a null pointer. Again, it's best not to break existing and shipping implementations without some very good reason. Third, Table 31 in the WP (the variable definition table, part of the allocator requirements description) implicitly assumes that a null pointer of type Allocator::pointer can be written as 0.

Arguments in favor of option 2. First, option 1 would usually mean that a user-defined pointer-like type would have a constructor from int or from void*. This might have undesirable implications. Second, Allocator::null is extremely clear. (Maybe it's so clear that it will finally be enough to end comp.lang.c++ null pointer wars.)

Arguments in favor of option 3. It doesn't require new syntax and doesn't require implicit conversions.

I favor either option 1 (because it is the smallest change to the current WP) or option 2 (because it s very explicit and clear). There was support within the library working group for each of these three options.

B. Preconditions and postconditions

This is a small nit. Tables 31 and 32 distinguish between pointers returned by Allocator::allocate(), and pointers that might be null. All that's missing is an explicit guarantee that allocate() never returns null pointers. (I think it's clear that this was everyone's intent. It's true for the default allocator, for example.)

Nobody in the library working group objected to this requirement.

C. Testing for null

If we have a pointer p, how do we test whether or not it's null? This is essentially a syntactic issue. The following are common ways of testing whether a pointer is null

```
    if (!p)
    if (p == 0)
    if (0 == p)
```

The following are common ways of testing whether a pointer is non-null.

```
    if (p)
    if (p != 0)
    if (0 != p)
```

(The expression "p == NULL" is not an additional form, since NULL is a macro that expands to a constant integral expression evaluating to zero.)

We need to say which of these, if any, are guaranteed to work. My preference is to provide all of them. Providing some, but not all, is just asking for trouble.

Also, of course, we can compare p to a value of type Allocator::pointer that is known to have a null value. In the event that we di decide to define a pointer constant Allocator::null, we won't have to do anything special in order to ensure that expressions like "p == Allocator::null" are well-defined.

Nobody in the library working group objected to providing a mechanism to test for null pointers.

III. CONVERSIONS TO OTHER USER-DEFINED POINTER TYPES

A. Conversion from Allocator::pointer to Allocator::const_pointer

This does not seem to be controversial. This should be an automatic conversion.

B. Conversion from Allocator::pointer to
   Allocator::rebind<void>::other::pointer.

Again, this should be an automatic conversion, and nobody in the library working group objected to it. Some people in the library working group proposed the generalization that Allocator::rebind<T>::other::pointer has an automatic conversion to Allocator::rebind<U>::other::pointer if T has an automatic conversion to U.

C. Conversion from Allocator::rebind<void>::other::pointer to
   Allocator::pointer.

This is clearly necessary. (Casting to void pointers is useless unless you can cast from void pointers.) Equally clearly, it shouldn't be an automatic conversion. Users should have to ask for it.

Issue: what should be the syntax for requesting this conversion? This is not a rhetorical question: I honestly don't have a good answer. I would like the answer to be static_cast<Allocator::pointer>(p), but I don't think that's possible. Unless I'm grossly misunderstanding the rules for static_cast (5.2.9), you can't use it for any sort of non-automatic user-defined conversions.

Option 1: invent some syntax (possibly involving a new static member function function of allocators) for this conversion.

Option 2: Simply declare, by fiat, that static_cast (or, equivalently, old-style C casts) must work. This is a restriction on Allocator, and anything that doesn't satisfy it is not a valid allocator.

Option 3: Don't allow this conversion at all: you can cast to a void pointer, but not from it. This would make void pointers effectively useless.

[Survey of existing practice. The SGI and HP implementations don't make any serious attempt to encapsulate alternative pointer types. The code in the Rogue Wave and Plum Hall implementations assumes Option 2. I don't know whether those implementations explicitly document this restriction.]

Nobody in the library working group proposed any additional options.

D. Conversion from Allocator::rebind<D>::other::pointer to
   Allocator::rebind<B>::other::pointer, where B is an
   unambiguous base class of D.

Again, derived-to-base conversion is clearly safe and useful. This ought to be an automatic conversion. (And it is useful when implementing container classes. The SGI implementation uses it in several different places. I suspect that the ObjectSpace implementation does too, but I haven't actually looked at their code.)

E. Downcasting. Conversion from Allocator::rebind<B>::other::pointer to Allocator::rebind<D>::other::pointer, where B is an unambiguous base class of D.

That is, p is of type pointer-to-B, but you, the programmer, happen to know that it actually a pointer to a D object. If p is a built-in pointer type, we can uses static_cast for this.

As in the case with section III.C, this is clearly a useful operation, it's allowed in the case of built-in pointers, but there doesn't seem to be a very good solution for providing it in the case of pointers that are user-defined types. The three options in III.C apply here too.

F. Other conversions.

The only other conversion that I've ever seen in container implementations is casting away constness. (The reason anyone cares about it is to avoid code duplication.) This is less important than downcasting or casting from void pointers, but it is still useful. Again, we have the same three options as in III.C and III.E.

IV. CONVERSIONS TO BUILT-IN POINTERS

The WP already provides a way of converting an Allocator<int>::pointer to a T*. We don't provide alternative reference types (they don't work), so, if p is of type Allocator<int>::pointer and q is of type Allocator<int>::const_pointer, then *p and *q are, respectively, of type int& and const int&. Then &*p and &*q are, respectively, of types int* and const int*. Of course, this only works if p and q are non-null.

The WP also provides a way of converting a built-in pointer to an Allocator::pointer. If A is an Allocator and p a non-null pointer of type Allocator::value_type*, then a.address(*p) has return type Allocator::pointer.

Two questions. First: do we need any other ways of performing these conversions? Second: some aspects of those conversions need to be described more precisely.

A. Other syntax

The existing syntax for conversions is slightly clumsy, but it's adequate. Conversions are uncommon.

Irrespective of syntax, this existing method has a minor disadvantage and a major flaw. The minor disadvantage is that it can't be used to convert null pointers. (The reason this is a relatively minor problem is that I can't find any place where any container really does need to convert a null Allocator::pointer to or from a null built-in pointer.)

The major flaw is that, although this method works for converting an Allocator<int>::pointer to an int*, it does not work for arbitrary types. If p is of type Allocator<T>::pointer, and T has user-defined operator& whose return type is U, then &*p will have type T* but rather will have type U. The only way to solve this problem would be to say that the standard library may not be used with type that overload operator&. This is probably an unacceptable restriction.

We need some other way to perform the conversion.

Option 1. Require that Allocator<T>::pointer and
Allocator<T>::const_pointer have implicit conversion to
T* and const T*.

Option 2. Invent some new syntax for the conversion, probably a
static member of Allocator.

Arguments in favor of option 1. First, it means we don't have to invent new syntax. Second, the pre-Kona WP said that there was an implicit conversions from Allocator::pointer to Allocator:value_type*. I don't know if there is any user code that relies on that conversion; we might want to consider providing it as an implicit conversions just to be safe.

Arguments in favor of option 2. Implicit conversions are always potentially dangerous, and we want to minimize them as much as possible.

I favor option 1, mainly because we should be cautious about inventing new syntax at this point. There was support within the library working group for both options.

B. Semantics of the conversion from X::pointer to T*.

Notation: X is an allocator type, T is X::value_type, x is an allocator, and p is a non-null X::pointer. The same issues apply to X::const_pointer. For the sake of the discussion in this section, I will assume that the static member function X::to_ptr converts an X::pointer to a T*. This notation is purely for the sake of clarity, and is not intended to prejudice the decision about whether conversion from X::pointer to T* should be by means of an implicit conversion or an explicit member function.

All of the issues in section IV.B are controversial.

1. Lifetime.

When we convert an X::pointer to a T*, what is the lifetime of the T* object? That is, consider the following.

```
  T* p1 = X::to_ptr(p);
  ++p;
```

The X::pointer, p, no longer points to the same thing. Is p1 still valid?

Option 1. p1 is valid so long as the object it points to is. The validity of p1 has nothing to do with the validity of p. It is valid until the object that it points to is destroyed.

Option 2. p1 has the lifetime of a temporary.

Option 3. p1 is valid until p is destroyed or made to point to something different.

Option 4. p1 is valid for the longer of the two periods in option 2 and option 3.

I'm fairly sure that strings require option 1. I think that option 4, and possibly option 3, would suffice for all other library components.

2. Identity

Is the expression X::to_ptr(p) == X::to_ptr(p) always guaranteed to be true? That is, under what circumstances do we always get the same T* when we do a conversion from an X::pointer?

Option 1: Yes. So long as the memory that p points into has not been deallocated, X::to_ptr(p) will always return the same T* value.

Option 2: No guarantees on whether or not p always refers to the same address.

Suggested resolution: option 1. My main reason for this preference is basically FUD: it's such a very fundamental property that I'm afraid there may be some code in the library (or elsewhere) that implicitly relies on this property even though that reliance isn't obvious.

3. Uniqueness

If p1 != p2, may we assume that X::to_ptr(p1) != X::to_ptr(p2)?

Same two options as in IV.B.2. I propose the same resolution, for the same reason. The idea that distinct objects have distinct address is so fundamental to C and C++ that I'm not sure what the consequences of a different rule would be.

(Again, this property is definitely necessary for strings. And, again, there aren't any other library components for which I'm sure that it's necessary.)

4. Conversions and arithmetic.

Suppose that p+n is a valid operation. Is it guaranteed that to_ptr(p) + n == to_ptr(p + n)? Similarly, is it guaranteed that to_ptr(p1) - to_ptr(p2) == p1 - p2? (John Skaller calls this property "structure-preserving conversion.)

Basic_string relies on this property. I'm reasonably sure that other library components don't.

5. Are strings a special case?

Basic_string relies on structure-preserving conversions, and probably also relies on identity and uniqueness. Three options.

Option 1. Guarantee structure-preserving conversions for all allocators.
Option 2. Document two different sorts of allocators. One kind isn't guaranteed to provide structure-preserving conversions, the other is.
Option 3. Don't allow alternate pointer types in strings at all.

I favor option 1. Option 2 would make for a more complicated standard (two allocator concepts instead of one) and it would invite some very horrible and hard-to-diagnose bugs. Option 3 would be acceptable too, but it seems unnecessarily restrictive. If user-defined pointer types are useful at all, then they're useful for strings.

There was support in the library working group for options 1 and 3. Nobody spoke in favor of option 2.

6. General discussion.

The properties in sections 1 through 4 are, clearly, closely related. I have proposed resolutions for all of them, but some members of the library working group disagree with those resolutions. The library working group did not reach consensus on these issues in Nashua.

Further analysis, possibly including implementation experience, is needed before we can reach a decision on these four issues.

C. Conversions from T* to X::pointer

The main question: if q is a non-null pointer of type T*, what are the preconditions for x.address(*q)? Table 32 has no preconditions at all, meaning that this operation is always valid.

Option 1: Remove the member function Allocator::address entirely.
Option 2: don't change table 32. Continue to have no preconditions on this operation.
Option 3: x::address(*q) is only permissible if q is a pointer that was obtained by converting a valid X::pointer to T*.
Option 4: similar to option 2, but also allow certain conversions between the X::pointer to T* and T* to X::pointer steps. For example, suppose that B is an unambiguous base class of D, pB is of type Allocator<B>::pointer, pB actually points to a D object, and aD is of type Allocator<D>. Should aD.address(static_cast<D*>(aB.to_ptr(pB))) be guaranteed to succeed?

Option 4 is an interesting possibility because it might provide an answer to the dilemmas of III.C, III.E, and III.F. It's an awfully ugly way to do something that ought to be simple, but maybe it's tolerable anyway.

Option 1 deserves consideration. Except for the purposes of conversion (which should probably be given clearer syntax in any case), it doesn't seem like a terribly useful member function. On the other hand, backwards compatibility is important too. This member function has been in the library for a long time, and it's awfully late to be talking about removing it.

Option 2 is probably not viable.

V. EXCEPTIONS

In my opinion, it's hopeless to talk about exception safety of library containers if valid operations on valid pointers can result in exceptions. (For details see X3J16/97-0019 = WG21/N0157.) We need to require that valid pointer operations on Allocator::pointer, Allocator::const_pointer, Allocator::size_type, and Allocator::difference_type can't result in exceptions. If we choose to retain the address() member function, we also need to guarantee that a.address(*p) can't throw an exception if p satisfies whatever validity rules we end up establishing.

(Note that Allocator::pointer might have some member functions unrelated to pointer arithmetic; I do not intend to rule out the possibility that those member functions might throw exceptions. Similarly, I do not intend to rule out the possibility of user-defined pointer types that use exceptions for range checking. We need wording in the standard that makes this clear.)

Similarly, we need to require that Allocator::~Allocator() and Allocator::deallocate() throw no exceptions, ever. Exceptions in destructors are bad news.