Doc No: N3235=11-0005

Date: 2011-02-23

Authors: Pablo Halpern
Intel Corp..
phalpern@halpernwightsoftware.com

Generalized pointer casts

Contents

National Body Comments and Issues

This paper proposes a new resolution for LWG issue 1289, which is part of NB comment US 2 to the July, 2010 FCD. It is also tangentially related to US 88.

Document Conventions

All section names and numbers are relative to the November 2010 WP, **N3225**.

> 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 Table 44 – Allocator requirements in [allocator.requirements] (section 20.2.5), of the current WP, the following conversion is required between `void_pointer` and `pointer`:

Table 44 – Allocator requirements

Expression	Return Type	Assertion/note pre-/post-condition	Default
`static_cast<X::pointer>(w)`	`X::pointer`	`static_cast<X::pointer>(w) == p`	

This requirement means that a pointer-like type must provide an explicit conversion constructor such that a pointer-to-`void` can be used to construct a pointer-to-`T`. LWG 1289 observes that "this explicit conversion weakens the safety of a smart pointer since the following expression (invalid for raw pointers) would become valid:

```
smart_ptr<void> smart_v = ...;
smart_ptr<T> smart_t(smart_v);
```

." In fact, it is quite difficult to build a pointer-like class that has both of the following two desirable properties yet still meets the requirements of an allocator pointer:

1. *Implicit* conversion from pointer-to-derived to pointer-to-base or from pointer-to-T to pointer-to-void.
2. *Explicit* conversion from pointer-to-base to pointer-to-derived or from pointer-to-void to pointer-to-T.

An implicit conversion constructor is required for the first property. Conversely, an `explicit` conversion constructor is desirable for the second property, if `static_cast` is to used for those conversions. The two conversion constructors would need to coexist, using SFINAE tricks to create non-overlapping overload sets. LWG 1289 recommends a cleaner solution: that the standard provide a `static_pointer_cast` function template as a user customization point instead of requiring the use of `static_cast` directly. The proposed resolution would use the same syntax for `static_pointer_cast` as currently exists for `shared_ptr`. See http://lwg.github.com/issues/lwg-closed.html#1289 for details.

LWG issue 1289 was moved to NAD Future at the October 2010 meeting in Batavia. At the time, it was seen as a good idea, but not essential for completing the standard. It was seen as a feature that could be added in the next round of standardization.

In this paper, I argue that this feature cannot easily be added to a later revision of the standard and that, therefore, it should be standardized in this round. The proposed wording in this paper differs from that in LWG 1289, but is conceptually similar and was developed in consultation with the author of LWG 1289, Ion Gaztañaga.

Argument against NAD future for LWG 1289

If the current `static_cast` requirement is changed to a `static_pointer_cast` in a future standard, it will create the same kind of breakage as was described in US 88. To recap in general terms the problem described in US 88: When a requirement is weakened, then any generic code written against the stronger requirement may fail to compile with a new type that does not conform to the older (stricter) requirement. In this case, code written to use `static_cast` with pointer-like types would not work with a pointer type that did not provide the necessary conversion constructor, even if such a pointer type meets the new requirement of a `static_pointer_cast` function. Although a standard-library implementation of `static_pointer_cast` that defers to `static_cast` would allow older pointer types to work with new container implementations, it would do nothing to allow the reverse.

Because of several mitigating factors, US 88 was closed as NAD. The problem described in LWG 1289, however, does not benefit from most of the mitigating factors that applied to US 88. Specifically:

1. **Mitigating Factor for US 88:** Because of various handicaps built into the C++03 definition of allocators, there are very few user-defined containers that currently depend on the full Allocator requirements.
   Does not apply to LWG 1289 because: We hope that the changes made in the allocator model in C++0x will make Allocators more popular, invalidating this argument in a few years.
2. **Mitigating Factor for US 88:** It is easy to create an adaptor that allows a C++0x-complaiant allocator to work with a container written to the C++0x specification.
   Does not apply to LWG 1289 because: Although it might be possible to create an adaptor for a pointer-like type, a related adaptor would also need to be created for the allocator type that uses it. This dual-adaptor idiom would not meet reasonable criteria for being considered *easy*.
3. **Mitigating Factor for US 88:** Adding extra members to an allocator so that it meets the requirements of both C++03 and C++0x does not create problems.
   Does not apply to LWG 1289 because: As described above, the conversion constructor needed to satisfy the current requirements is not just inconvenient, but *undesirable*; thus users would be discouraged from creating robust pointer-like types if they want to remain compatible with both C++0x and a future standard that supported `static_pointer_cast`. This factor is probably the most problematic aspect of the issue.
4. **Mitigating Factor for US 88:** The "fix" to US 88 would do damage to the simplicity of the C++0x allocator model.

Does not apply to LWG 1289 because: The fix to LWG 1289 described in this proposal is not onerous or disruptive to the allocator model. It is, in fact, very much in keeping with the philosophy of using traits classes (`allocator_traits` and `pointer_traits`) as adaptation points.

I assert that the absence of cast functions in `pointer_traits` was an oversight and is a defect; they belong there as a way to maintain flexibility for the future.

Summary of Proposed Changes

This proposal uses a slightly different approach than the proposed resolution in LWG 1289. LWG 1289 would replace `static_cast<X::pointer>(w)` with a customizable (via ADL) `static_pointer_cast<X::value_type>(w)`. Instead of using ADL, this paper proposes that the customization be made by adding a `static_pointer_cast` member to `pointer_traits`. For convenience, a namespace-scoped `std::static_pointer_cast` calls the version in `pointer_traits`. I.e.,

```
namespace std {
  template <class Ptr>
  struct pointer_traits {
    ...
    template <class U>
      static rebind<U> static_pointer_cast(const Ptr& p);
  };

  template <class T, class Ptr>
  auto static_pointer_cast(Ptr&& p) -> typename
    pointer_traits<typename std::decay<Ptr>::type>::template rebind<T>
  {
    return pointer_traits<typename std::decay<Ptr>::type>::
        static_pointer_cast<T>(std::forward<Ptr>(p));
  }
}
```

This approach has several advantages over the ADL mechanism:

- It centralizes all pointer-related customizations in the `pointer_traits` class.
- It prevents ADL-related issues such as those potential ambiguities plaguing the `begin()` and `end()` namespace-scoped functions.
- It avoids requiring a "`using std::static_pointer_cast;`" declaration at every use. (I have found this to be a nuisance with `swap`.)

For symmetry, we also add `const_pointer_cast` and `dynamic_pointer_cast`. Note that a particular specialization of `pointer_traits` need not supply all three casts. Only `static_pointer_cast` from `void_pointer` is required for an allocator's `pointer` type. (An additional proposal at the end of this paper would add `const_pointer_cast` to the list of requirements.)

For consistency, a `pointer_traits` structure was added for `shared_ptr`. The `shared_ptr` casts have been replaced with the corresponding generic templates, but no change in syntax or semantics should result.

Implementation Experience

Implementing the facilities described in this proposal is quite simple. A full implementation of `pointer_traits` with a test driver (tested on using gcc 4.4.3) is available at http://www.halpernwightsoftware.com/WG21/allocator_traits_n3235.tgz. This archive also includes an implementation of `allocator_traits`, `scoped_allocator_adaptor` and `std::list` using `pointer_traits`.

Proposed Wording

In Section 20.2.5 [allocator.requirements] Table 44, change the `static_cast` requirement:

Table 44 – Allocator requirements

Expression	Return Type	Assertion/note pre-/post-condition	Default
~~static_cast<X::pointer>(w)~~ `std::static_pointer_cast<T>(w)`	`X::pointer`	~~static_cast<X::pointer>(w)~~ `std::static_pointer_cast<T>(w) == p`	
~~static_cast<X::const_pointer>(z)~~ `std::static_pointer_cast<const T>(z)`	`X::const_pointer`	~~static_cast<X::const_pointer>(z)~~ `std::static_pointer_cast<const T>(z) == q`	

In section 20.9 [memory], add three function templates to the synopsis:

```
// 20.9.3, pointer traits
template <class Ptr> struct pointer_traits;
template <class T> struct pointer_traits<T*>;

// 20.9.x pointer casts
template <class T, class Ptr>
  rbptr static_pointer_cast(Ptr&& p) noexcept;
template <class T, class Ptr>
  rbptr const_pointer_cast(Ptr&& p) noexcept;
template <class T, class Ptr>
  rbptr dynamic_pointer_cast(Ptr&& p) noexcept;
```

And (also in 20.9 [memory]), replace the shared pointer casts with shared pointer traits:

// 20.9.10.2.10, shared_ptr ~~casts~~pointer traits:

~~`template<class T, class U>`~~

~~`  shared_ptr<T> static_pointer_cast(shared_ptr<U> const& r);`~~

~~`template<class T, class U>`~~

~~`  shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const& r);`~~

~~`template<class T, class U>`~~

~~`  shared_ptr<T> const_pointer_cast(shared_ptr<U> const& r);`~~

`template <class T> struct pointer_traits<shared_ptr<T>>;`

In section 20.9.3 [pointer.traits], add some text and add three function templates to `pointer_traits`:

The class template `pointer_traits` supplies a uniform interface to certain attributes of pointer-like types. A user may define a specialization for `std::pointer_traits` for a user-defined pointer-like class. Such a specialization shall contain definitions of the `pointer` and `element_type` members, but is not required to define every member of the primary template. [*Note:* Other parts of this standard impose additional constraints on a specialization. For example, according to the Allocator requirements (20.2.5), the `pointer` type for an Allocator requires the presence of an `std::static_pointer_cast` function that creates a `pointer` from a `corresponding void_pointer`. This function, in turn, requires that `pointer_traits<pointer>::static_pointer_cast` be defined to implement this conversion. – *end note*]

```
namespace std {
  template <class Ptr> struct pointer_traits {
    typedef Ptr pointer;
    typedef see below element_type;
    typedef see below difference_type;

    template <class U> using rebind = see below;

    static pointer pointer_to(see below r);

    template <class U>
      static rebind<U> static_pointer_cast(const pointer& p) noexcept;
    template <class U>
      static rebind<U> const_pointer_cast(const pointer& p) noexcept;
    template <class U>
      static rebind<U> dynamic_pointer_cast(const pointer& p) noexcept;
  };

  template <class T> struct pointer_traits<T*> {
    typedef T element_type;
    typedef T* pointer;
    typedef ptrdiff_t difference_type;

    template <class U> using rebind = U*;

    static pointer pointer_to(see below r);

    template <class U> static U* static_pointer_cast(T* p) noexcept;
    template <class U> static U* const_pointer_cast(T* p) noexcept;
    template <class U> static U* dynamic_pointer_cast(T* p) noexcept;
  };

  template <class Ptr> struct pointer_traits<const Ptr>;
  template <class Ptr> struct pointer_traits<volatile Ptr>;
  template <class Ptr> struct pointer_traits<const volatile Ptr>;
}
```

Each member of a specialization of `pointer_traits` on a *cv*-qualified type `cv Ptr` shall be identical to the corresponding member of the specialization on the unqualified type `Ptr`.

Add the following to the end of section 20.9.3.2 [pointer.traits.functions]:

```
template <class U>
  static rebind<U> static_pointer_cast(const pointer& p) noexcept;
template <class U>
  static U* pointer_traits<T*>::static_pointer_cast(T* p) noexcept;
```

Requires: The expression `static_cast<U*>(declval<element_type*>())` shall be well formed (in an unevaluated context). For the first template, if the expression `p.template static_pointer_cast<U>()` is well-formed then the result of that expression shall be implicitly convertible to `rebind<U>`.

Returns: The first template returns the result of calling `p.template static_pointer_cast<U>()` if such an expression is well-formed; otherwise, it returns `static_cast<rebind<U>>(p)` if such an expression is well formed; otherwise the specialization is ill-formed. The second template returns `static_cast<U*>(p)`.

```
template <class U>
  static rebind<U> const_pointer_cast(const pointer& p) noexcept;
template <class U>
  static U* pointer_traits<T*>::const_pointer_cast(T* p) noexcept;
```

Requires: The expression `const_cast<U*>(declval<element_type*>())` shall be well formed (in an unevaluated context). For the first template, if the expression `p.template const_pointer_cast<U>()` is well-formed then the result of that expression shall be implicitly convertible to `rebind<U>`.

Returns: The first template returns the result of calling `p.template const_pointer_cast<U>()` if such an expression is well-formed; otherwise the specialization is ill-formed. The second template returns `const_cast<U*>(p)`.

```
template <class U>
  static rebind<U> dynamic_pointer_cast(const pointer& p) noexcept;
template <class U>
  static U* pointer_traits<T*>::dynamic_pointer_cast(T* p) noexcept;
```

Requires: The expression `dynamic_cast<U*>(declval<element_type*>())` shall be well formed (in an unevaluated context). For the first template, if the expression `p.template dynamic_pointer_cast<U>()` is well-formed then the result of that expression shall be implicitly convertible to `rebind<U>`.

Returns: The first template returns the result of calling `p.template dynamic_pointer_cast<U>()` if such an expression is well-formed; otherwise the specialization is ill-formed. The second template returns `dynamic_cast<U*>(p)`.

Between section 20.9.3 [pointer.traits] and section 20.9.4 [allocator.traits], add a new section:

20.9.x pointer casts [pointer.cast]

In the function descriptions that follow, let *rbptr* be `typename pointer_traits<typename decay<Ptr>::type>::template rebind<T>`.

```
template <class T, class Ptr>
  rbptr static_pointer_cast(Ptr&& p) noexcept;
```

Returns: `pointer_traits<typename decay<Ptr>::type>::static_pointer_cast<T>(std::forward<Ptr>(p)).`

```
template <class T, class Ptr>
  rbptr const_pointer_cast(Ptr&& p) noexcept;
```

Returns: `pointer_traits<typename decay<Ptr>::type>::const_pointer_cast<T>(std::forward<Ptr>(p)).`

```
template <class T, class Ptr>
  rbptr dynamic_pointer_cast(Ptr&& p) noexcept;
```

Returns: `pointer_traits<typename decay<Ptr>::type>::dynamic_pointer_cast<T>(std::forward<Ptr>(p)).`

In section 20.9.10.2.9 [util.smartptr.shared.cast], replace the shared pointer cast free-functions with shared pointer traits:

20.9.10.2.10 shared_ptr ~~casts~~pointer traits [util.smartptr.shared.~~cast~~ptrtrait]

```
namespace std {
  template <class T>
  struct pointer_traits<shared_ptr<T>> {
    typedef shared_ptr<T> pointer;
    typedef T             element_type;
    typedef ptrdiff_t     difference_type;

    template <class U> using rebind = shared_ptr<U>;

    static pointer pointer_to(unspecified r) = delete;

    template <class U>
      static shared_ptr<U> static_pointer_cast(const shared_ptr<T>& p) noexcept;
    template <class U>
      static shared_ptr<U> dynamic_pointer_cast(const shared_ptr<T>& p) noexcept;
    template <class U>
      static shared_ptr<U> const_pointer_cast(const shared_ptr<T>& p) noexcept;
  };
}
```

20.9.10.2.10.1 shared_ptr traits members

`template<`~~`class T, `~~`class U>`
`  static shared_ptr<`~~`T`~~`U> static_pointer_cast(const shared_ptr<`~~`U`~~`T>& r) noexcept;`

1 *Requires*: The expression `static_cast<`~~`T`~~`U*>(r.get())` shall be well formed.

2 *Returns*: If `r` is empty, an empty `shared_ptr<`~~`T`~~`U>`; otherwise, a `shared_ptr<`~~`T`~~`U>` object that stores `static_cast<`~~`T`~~`U*>(r.get())` and shares ownership with r.

3 *Postconditions*: `w.get() == static_cast<`~~`T`~~`U*>(r.get())` and `w.use_count() == r.use_count()`, where `w` is the return value.

4 [*Note*: The seemingly equivalent expression shared_ptr<~~T~~U>(static_cast<~~T~~U*>(r.get())) will eventually result in undefined behavior, attempting to delete the same object twice. —*end note*]

template<~~class T, ~~class U>
  static shared_ptr<~~T~~U> dynamic_pointer_cast(const shared_ptr<~~U~~T>& r) noexcept;

5 Requires: The expression dynamic_cast<~~T~~U*>(r.get()) shall be well formed and shall have well defined behavior.

6 *Returns*:

— When dynamic_cast<~~T~~U*>(r.get()) returns a nonzero value, a shared_ptr<~~T~~U> object that stores a copy of it and shares ownership with r;

— Otherwise, an empty shared_ptr<~~T~~U> object.

7 *Postcondition*: w.get() == dynamic_cast<~~T~~U*>(r.get()), where w is the return value.

8 [*Note*: The seemingly equivalent expression shared_ptr<~~T~~U>(dynamic_cast<~~T~~U*>(r.get())) will eventually result in undefined behavior, attempting to delete the same object twice. —*end note*]

template<~~class T, ~~class U>
  static shared_ptr<~~T~~U> const_pointer_cast(const shared_ptr<~~U~~T>& r) noexcept;

9 *Requires*: The expression const_cast<~~T~~U*>(r.get()) shall be well formed.

10 *Returns*: If r is empty, an empty shared_ptr<~~T~~U>; otherwise, a shared_ptr<~~T~~U> object that stores const_cast<~~T~~U*>(r.get()) and shares ownership with r.

11 *Postconditions*: w.get() == const_cast<~~T~~U*>(r.get()) and w.use_count() == r.use_count(), where w is the return value.

Additional Proposal: Requiring const_pointer_cast

A survey by Ion Gaztañaga of existing standard and Boost container implementations shows that it may also be a good idea to require const_pointer_cast as part of the allocator requirements. I propose this addition here as a separable proposal, since it is related to, but goes beyond, the original issue and thus might be considered out of scope.

Proposed wording

In Section 20.2.5 [allocator.requirements] Table 44, add the following two rows:

Table 44 – Allocator requirements

Expression	Return Type	Assertion/note pre-/post-condition	Default
std::const_pointer_cast<T>(q)	X::pointer	std::const_pointer_cast<T>(q) == p	
std::const_pointer_cast<void>(z)	X::void_pointer	std::const_pointer_cast<void>(z) == w	

Acknowledgements

Thanks to Ion Gaztañaga, Alisdair Meredith, Mike Giroux, John Lakos and especially Daniel Krugler for their input and review.

References

N3102: ISO/IEC FCD 14882, C++0X, National Body Comments
LWG 1289: Generic casting requirements for smart pointers