Document number: N4166
Date: 2014–10–06
Reply to: David Krauss
(david_work at me dot com)

Movable initializer lists

1. Abstract

Often `std::initializer_list` cannot be used, or it requires a `const_cast` hack, as it provides read-only access to its sequence. This is a consequence of the sequence potentially being shared with other `initializer_list` objects, although often it is statically known to be uniquely local. A new initializer list class template is proposed to allow function parameters which may leverage (by overloading) or require strict ownership. Addition of this class does not impinge on the cases where the sequence should be shared. No breaking changes are proposed.

2. Background

`std::initializer_list` was designed around 2005 (N1890) to 2007 (N2215), before move semantics matured, around 2009 [1]. At the time, it was not anticipated that copy semantics would be insufficient or even suboptimal for common value-like classes. There was a 2008 proposal N2801 *Initializer lists and move semantics* but C++0x was already felt to be slipping at that time, and by 2011 the case had gone cold. It shares many similarities with this proposal.

Ownership is very important in modern C++. Users often specify move semantics in the course of an idiomatic pattern, not for the sake of performance. In particular, `std::unique_ptr` is popular in general use for its simplicity and safety. Although `std::initializer_list` is often used in constructors to specify the content of a container, it does not own its sequence and cannot give permission to modify, never mind assume ownership of sequence elements. Instead, its interface is based on a generalization where the sequence may be initialized at program startup, and different lists may alias the same immutable sequence. This model is impossible to realize, however, if any list element depends on the context of initialization. Other aspects of object initialization require constructors and destructors to be called, further limiting this optimization. In the majority of uses, ownership exists but it is hidden from the user.

When every list element depends on a value computed in its scope, or the list is only used once during the entire program execution and the elements are not of literal type, ownership logically must exist. In these cases, it is safe to remove the `const` by a `const_cast` in order to complete a move operation. This is hardly an acceptable practice, and few users will extrapolate the rules correctly. The language needs a facility to safely expose a non-`const` underlying sequence.

---

[1] To be fair, `move()` was first formally presented in 2002 (N1377), and in a mature form, but it remained "in the laboratory" for quite a while. Also, before the appearance of `initializer_list`, proposals since 2003 had applied array objects directly to the same initialization problem.

3. Proposal

A class derived from `std::initializer_list<T>` is proposed, as ownership is a superset of observation. Since it implements all the same members, it is provided as a specialization of the `std::initializer_list` template. However, it additionally implements an empty moved-from state, and it assumes ownership in that its destructor destroys the sequence. The iterator type is a non-`const` pointer, so that the user may apply `std::move` to its elements.

```
template< typename T >
struct initializer_list< T && >
    : initializer_list< T > {
    typedef T & reference;
    typedef T * iterator;

    using initializer_list< T >::begin;
    constexpr iterator begin() noexcept;
    using initializer_list< T >::end;
    constexpr iterator end() noexcept;

    constexpr initializer_list();
    initializer_list( initializer_list const & ) = delete;
    constexpr initializer_list( initializer_list && );

    ~ initializer_list()
        noexcept(noexcept(begin()->~T()));
};
```

A *braced-init-list* may be passed to an overloaded function, where the corresponding parameter types include both `initializer_list<T&&>` and `initializer_list<T>`. Rather than initialize these objects directly by copy-list-initialization ([dcl.init.list] §8.5.4 [2]), which would result in overload ambiguity, the list is used to generate a prvalue expression of one of these two types, and that is used as an argument. An owning list (`<T&&>`) will initialize an observing list (`<T>`), but prefer to confer ownership, by the derived-to-base conversion. An observing list will only initialize another observing list. The temporary object associated with the prvalue expression may be elided ([class.copy] §12.8/31). Whichever overload is selected, the behavior is still the same as copy-list-initialization of the parameter.

When the implementation generates a temporary initializer list object from a *braced-init-list*, that object is a specialization of `initializer_list<T&&>` if the the underlying array is not `const`, and its lifetime coincides with that of the temporary list object. These decisions are at the implementation's discretion, for the sake of optimization. However, to prevent arbitrary overload failure, an owning temporary object must be generated if no corresponding non-owning parameter is present in the overload set.

The rvalue reference qualifier `&&` in the template argument has no effect except to select the specialization. Currently, `initializer_list` specializations on reference type are ill-formed

[2] References are to the C++14 FCD, N3936 unless otherwise noted.

because the `iterator` member declaration would form a pointer to reference type. When the initializer list type is deduced per [temp.deduct.call] §14.8.2.1, only the type qualified by an rvalue reference modifier is deduced. (*P'* in the specification is treated as *P'* `&&`$_{opt}$, and *P'* itself will not be an rvalue reference type. The case of lvalue references does not matter, as that will produce an ill-formed specialization.) Generically deducing the type and ownership of an initializer list requires two template overloads. A local variable declared as `auto` and initialized with a *braced-init-list* will never expose a writable sequence.

Assignability of all `initializer_list` specializations must be forbidden. C++14 seems to ambiguously permit assignability, contingent on the implementation using non-`const` data members. The semantics are intrinsically broken and easily result in dangling pointers. Changing the base, observing-only subobject of an owning object would be disastrous, so this is a good opportunity to completely stamp out `initializer_list::operator =` . This issue has also been filed as LWG DR 2432.

4. Examples

Ordinary `initializer_list<T>` continues to work as usual.

```
void a( std::initializer_list< int > seq ) { // #1
    for ( auto && i : seq ) {
        i = 5; // error: i has type int const &.
    }
}

a({ 1, 2, 3 }); // Sequence probably has static storage.
a({ errno }); // Sequence probably has automatic storage.
```

When an `initializer_list<T&&>` overload is added to the mix, it handles lists with exclusive access to the sequence.

```
void a( std::initializer_list< int && > seq ) { // #2 (overload)
    for ( auto && i : seq ) {
        i = 5; // OK. (In practice you would move something.)
    }
}

a({ 1, 2, 3 }); // Probably calls #1.
a({ errno }); // Probably calls #2.
```

Such a function may usually be implemented generically:

```
template< typename t > // t is foo or foo &&.
void generic( std::initializer_list< t > seq ) {
    for ( auto && f : seq ) { // f is const or modifiable.
        smth( std::move( f ) ); // Move is defeated by const.
    }
}
```

```
void a( std::initializer_list< foo > seq )
    { generic( seq ); }
void a( std::initializer_list< foo && > seq )
    { generic( seq ); }
```

If there is only an `initializer_list<T&&>` overload, the list is required to own the sequence.

```
void b( std::initializer_list< std::unique_ptr<int> && > seq );

b({ std::make_unique< int >( 1 ), nullptr }); // OK
```

When the template parameter of `initializer_list` is deduced from the content of the list, it the ownership status is not deduced but an `&&` modifier does not interfere with deduction.

```
template< typename T >
void c( std::initializer_list< T > seq ); // #3

c({ 1, 2, 3 }); // T = int, no write access.
c({ errno }); // Also T = int, no write access.

template< typename T >
void d( std::initializer_list< T && > seq );

d({ 1, 2, 3 }); // T = int, write access is guaranteed.

template< typename T >
void c( std::initializer_list< T && > seq ); // #4 (overload)

c({ errno }); // Probably calls #4.
```

The behavior of the `auto x = { … }` syntax is unchanged; such a variable never exposes write access.

```
auto e = { errno };
* e.begin() = 5; // Error: begin() has type int const *.

for ( auto && p : { std::make_unique< int >( 5 ) } ) {
    foo( std::move( p ) ); // Error: p is constant.
}
for ( auto && p                   // Explicitly specify ownership:
          : std::initializer_list< std::unique_ptr< int > && >
              { std::make_unique< int >( 5 ) } ) {
    foo( std::move( p ) ); // OK
}
```

When the user knows there is no benefit to sharing, explicitly naming the template with an rvalue reference type argument guarantees move semantics. This template-id resembles the form of `static_cast< T && >` used to accomplish a move.

```
std::map< foo, bar > global_table
    = std::initializer_list< std::pair< foo, bar > && > {
    { "lala", { 54 } }, { "barf", { 33 } }
};
```

5. Rationale

A partial specialization is chosen over a new primary template for the sake of familiarity and simplicity. The `&&` qualifier should be about as easy to remember as adding `movable` to the template name, and both have similar connotations. Typical users should not need to be aware of the subclass relationship in particular, but it should be intuitive in any case that an owner object may initialize an observer object. This may be more obvious to the user from a similar class name, even if differently-specialized templates are unrelated types in general. Less importantly, much of the current specification is hard-coded to the `initializer_list<T>` name, and avoiding introduction of `movable_initializer_list` reduces the required normative changes. Finally, sharing the primary template enables generic functions which accept owning or non-owning lists. In the generic context, if `iterator` is `T const *`, then `move(*iter)` will yield `T const &&`, which tends to behave just like `T const &`, i.e. move semantics applied to a read-only list are converted into pure observation. Function overloads accepting owning and observing lists can funnel into such a function template, which would not be possible if the new class were not an `initializer_list` specialization.

Partial specialization on a distinct type is chosen over partial specialization with SFINAE discrimination by a metafunction (such as `is_literal`) because the behavioral variation of `initializer_list` is based not on its type parameter, but on the implementation's choice of underlying storage per sequence. Also, in the general case, literal types may have distinct copy and move semantics, and moving is presumably less costly. There is already an effort to unify compile-time and runtime string classes, so `std::vector<string>` could very conceivably be tasked with accepting either string objects in ROM or temporary strings residing on the stack. As a matter of evolution, the requirements for literal and ROMable types will tend to relax. Switching such types to copy-only semantics would produce breakage, and generate resistance to expanding the scope of these positive qualities in the language.

Unique ownership and move semantics are chosen over potentially-shared, read-write access to the sequence for the sake of efficiency and discouraging inappropriate usage. Easily shared write access would encourage usage as algorithm scratch space, which is against the basic intent of `initializer_list`. Ownership allows resources not freed by move-initialization to be freed as soon as the owning list is destroyed. Moreover, each list object that assumes ownership of the sequence will be destroyed sooner than the previous owner, following the principle that destruction occurs in the order opposite from construction. Ownership inclusive of destruction also eliminates the messy question of the underlying array's status as an independent object with its own lifetime. (Does an array bound to a parameter object really need to persist after the function call until the end of the full-expression, as specified by [dcl.init.list] §8.5.4/9?) The array is demoted from a pseudo-temporary object to a storage space. Compatibility is unaffected as the lifetime specified by C++14 still applies unless the user declares a `<T&&>` specialization.

Inheritance is chosen over a user-defined conversion or completely distinct classes because it is the most elegant solution. Inheritance from a specialization of the same template is somewhat unusual, but not particularly so. The alternatives amount to workarounds with special cases added to overload resolution, or implicit function calls that need removal by optimization.

Read-write access requiring an explicit call to `move` per element is chosen over behavior like `move_iterator` for safety and general sensibility. Although `initializer_list` is not a container and one pass of iteration should usually suffice, there is no need to impede observation of an element before deciding where to transfer it.

Ownership is specified, not deduced, when the list element type is deduced from the items in the initializer list, to avoid foisting modifiability and deduction of rvalue reference types on generic functions. This is also the most straightforward way to modify the existing specification, and it still provides for discriminating ownership in the same way as non-template overloads. The same applies to `auto` local variables. If they had deduced ownership, it would lead to surprisingly implementation-dependent `const`ness, especially in range-for loops as in the final example.

This proposal makes it easier to observe the implementation-specific behavior of list storage strategy, but this is acceptable and comparable to another observable optimization, copy elision. In both cases, the implementation chooses to combine the identity of objects that would otherwise be distinct. The user must always opt-in to this proposal's behavioral variance by supplying multiple overloads. This is a safe solution, and preferable to over-specification which may forbid optimizations.

6. Future work

No prototype yet exists. Practice makes perfect.

In the standard library, initializer list constructors of class templates supporting element move semantics will need overloads accepting sequence ownership. This includes the proper containers but not `valarray` or `string`. A cursory search found other uses of `initializer_list` in the algorithm and regular expression libraries, but nothing requiring adjustment.

Some convenient syntax may be invented to request sequence ownership in a range-for loop.

ROM-friendly, shared-object behavior would be nice to have for all prvalues, not only those used to initialize `initializer_list`.

7. Acknowledgements

Rodrigo Campos, the author of N2801, provided helpful feedback and kind encouragement.

Ville Voutilainen provided valuable guidance and review.