Document number:	P0065
Date:	2015–09–27
To:	SC22/WG21 EWG
Reply to:	David Krauss (david_work at me dot com)
Revises:	N4166

Movable initializer lists, rev. 2

Elements must be copied, not moved, from `std::initializer_list`, as it provides read-only "view" access to a `const`-qualified sequence. This causes a loss of performance and generality which often renders it unusable. A new template `own_initializer_list` is proposed to provide proper ownership of the initializer list sequence. It works like a move-only, fixed-length, stack-allocated `std::vector` under an `initializer_list`-like class interface.

1. Background

`std::initializer_list` was introduced as part of the "generalized initializer lists" feature [1], with the intent of providing an abstraction for aggregate initialization. From N2215 (Stroustrup 2007, emphasis added):

> "… we propose:
>
> - To allow an initializer list (e.g., `{1,2,3}` or `={1,2,3}`) wherever an initializer can appear (incl. as a return expression, an function argument, a base or member initializer, and an initializer for an object created using `new`). **An initializer list appears to the programmer as an rvalue**.
> - To introduce type `std::initializer_list` for the programmer **to use as an argument type when an initializer list is to be accepted** as an argument. …

The status quo falls short of this goal because it does not support move semantics, which developed concurrently (i.e., over the entire C++11 gestation) but independently. N2719 and N2801 (Campos 2008) attempted unification, but they were considered officially "too late" for the slipping C++0x.

In 2011, move-only types seemed exotic, and deep copies were the status quo. Today, `std::unique_ptr` is a vocabulary type and it's surprising if anything is needlessly copied. When a sequence of move-only function parameters is needed, several alternatives do exist, such as perfect forwarding, `std::vector` as an initializer list, arrays with move iterators, custom reference wrappers, and illegal `const_casting`. These workarounds range from inefficient to inconvenient to embarrassing. The preponderance of alternatives leads to inconsistency.

Compared to its predecessor N4166, this proposal renames the partial specialization as a new class template, and fully elaborates the overloading rules and implementation details. Deduction by `auto` has been changed to support and reflect ownership. The text has been rewritten.

[1] N2215, *Initializer lists (rev. 3)*, N1493 *Braces initialization overloading*, N1509 *Generalized initializer lists*, many other proposals, http://engineering.tamu.edu/media/728036/IAP_Initializers.pdf, etc.

2. Proposal

A new class template is proposed, `std::own_initializer_list<T>`, implementing ownership semantics similar to `std::vector` but using stack allocation. This class is derived from `std::initializer_list<T>`, as ownership is a superset of observation. It implements an empty moved-from state, and 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 own_initializer_list
    : initializer_list< T > {
    typedef T & reference;          // Inherit const_iterator, const_reference.
    typedef T * iterator;

    constexpr iterator begin() noexcept;
    constexpr const_iterator begin() const noexcept;          // As inherited.
    constexpr iterator end() noexcept;
    constexpr const_iterator end() const noexcept;            // As inherited.

    constexpr own_initializer_list() noexcept;
    own_initializer_list
        ( own_initializer_list const & ) = delete;
    constexpr own_initializer_list
        ( own_initializer_list && ) noexcept;          // Post: size() == 0.

    ~ own_initializer_list();
};
```

A *braced-init-list* may initialize ([dcl.init.list] §8.5.4 [2]) an `own_initializer_list<T>` as it would an `initializer_list<T>`, including the case where `T` is deduced. For a given `T`, one of these two classes is preferred, depending on whether `own_initializer_list` enables useful move semantics. Specifically, `own_initializer_list<T>` is preferred if and only if `T` has a nontrivial move constructor `T::T(T&&)`. Most classes with a nontrivial move constructor also have a nontrivial destructor (according to the "rule of five"), disqualifying them from the static sequence storage optimization implemented by `initializer_list`. Note that a class defining a copy constructor but no move constructor does not have a nontrivial move constructor; it has no move constructor at all, so the preferred class remains `initializer_list<T>`. Likewise, for const-qualified `T`, the relevant move constructor is `T::T(T const &&)`, which seldom exists, so `initializer_list<T>` will probably be chosen.

In function overloading, the conversion sequence representing initialization of `own_initializer_list<T>` is a list-initialization sequence ([over.ics.list] §13.3.3.1.5/4). Overload ambiguity is avoided by treating conversion to the preferred initializer list type as a better conversion sequence in [over.ics.rank] §13.3.3.2/3.1, given a choice between two initializer list types of the same `T` [3].

[2] References are to the working draft N4527 unless otherwise noted.

[3] It may be useful to define *initializer list types of `T`* as a concept, perhaps including array types.

The non-preferred type may still be used explicitly. Therefore it is not appropriate to provide an `own_initializer_list` overload without an `initializer_list` alternative, unless the function will move from the sequence and the type is not copyable. The simplest approach for container libraries is to always provide both overloads. Overloading `initializer_list` against `own_initializer_list &&` allows an lvalue `own_initializer_list` argument to select the `initializer_list` overload by the derived-to-base conversion instead of accessing the deleted copy constructor. Non-destructive algorithms such as `std::min` and `std::max` are not affected by this proposal. (This proposal does not address the standard library, but it is covered by the prototype.)

The preferred initializer list type is used when deducing `auto` from an initializer list. Whereas [dcl.spec.auto] §7.1.6.4/7 currently says to pass the initializer list to an imaginary function template with parameter type `initializer_list<U>`, the preferred type may be resolved by passing it to an overload set including a parameter of type `own_initializer_list<U>`.

Assignability of `initializer_list` and `own_initializer_list` specializations must be forbidden. C++14 leaves assignment unspecified. In practice it is allowed, but it only works when the static sequence storage optimization applies (i.e., the sequence is trivially destructible, the list initializers are all constant expressions, and the implementation chooses to use static storage duration for the underlying array). Otherwise, the status quo is to assign dangling pointers. Calling `std::own_initializer_list::initializer_list::operator=` would lead to double deletion, so this is a good opportunity to completely stamp out assignment altogether. This issue has also been filed as LWG DR 2432.

To support constant expressions, `own_initializer_list<T>` is a literal type when `T` is a literal type. Specifically, the destructor of `own_initializer_list<T>` is ignored when `T` is trivially destructible. The implementation could accomplish this, for example, by specializing `std::is_trivially_destructible`. However, because initializer list types should support incomplete sequence types (see LWG DR 2493), it is not appropriate to use partial specialization to select a trivial destructor: Selecting the specialization would require `T` to be complete.

For the container to destroy the sequence, their lifetimes must end at the same time. Therefore, `own_initializer_list` (or array types formed of it) are forbidden in cases where sequence lifetime extension would fail: when used as the value of a `return` statement, as an exception object, in a class definition as a subobject (i.e., as a base class or a nonstatic member), or as the type in a *new-expression*. These restrictions do not apply to `initializer_list`, nor to reference types to `own_initializer_list` where they would be applicable.

3. Examples

The implementation's choice between the owning, modifiable `own_initializer_list`, or the observing, read-only `initializer_list`, is based on the element type.

```
auto li = { 1, 2, 3 };          // li has type std::initializer_list< int >.
auto ls = { std::string( "hello" ) };
                    // ls  has type std::own_initializer_list< std::string >.
auto li2 = li;                              // OK: initializer_list is copyable.
auto ls2 = ls;                       // Error: own_initializer_list is not copyable.
auto ls3 = std::move( ls );               // OK: own_initializer_list is movable.
```

The user passes ownership through a function call by moving the list object. Read-only access still applies in the unusual case that the list is named as an lvalue, if the function supports this.

```
std::vector< std::string > vsc = ls; // OK: slice to initializer_list and copy.
std::vector< std::string > vsm = std::move( ls );   // OK: move from sequence.
assert ( ls.size() == 0 );                       // Moved-from sequence no longer exists.
std::vector< std::string > vs = { "yizzo" };        // OK: move from sequence.
std::vector< int > vim = std::move( li );          // OK: move devolves to copy.
std::vector< int > vi = { 1, 2, 3 };               // OK: move devolves to copy.
```

Algorithms iterating over an initializer list, including range-based `for` loops, may presume `std::move` to be safe. If the list is not owning, then the `iterator` type is `T const *`, and moving an element yields `T const &&`, which is safe (and typically just like `T const &`).

```
template< typename value, typename container >
void distribute( value const & v, container & c ) {
    auto it = c.begin();
    // Transform v by functions foo, bar, and baz. Append results to initial elements of c.
    for ( auto && t : { foo( v ), bar( v ), baz( v ) } )
        it ++ ->push_back( std::move( t ) );      // Harmless for immovable t.
}
```

3.1. Container classes

A pass-by-value `own_initializer_list` overload supports move-initialization from a *braced-init-list*. For better or worse, this does not support copy-initialization from a named object of type `own_initializer_list`.

```
class container {
    container( std::initializer_list< std::string > il )
        : container( il.begin(), il.end() ) {}       // Delegating constructor.
    container( std::own_initializer_list< std::string > il )
        : container( std::make_move_iterator( il.begin() ),
                     std::make_move_iterator( il.end() ) ) {}
    // Iterator-based constructors, etc.
};
container a = { "hello", "world" };                              // OK
auto il = { std::string{ "hello" }, std::string{ "world" } };
container b = il;                              // Error: owning list passed as lvalue.
```

To copy when an `own_initializer_list` is passed as an lvalue, instead provide an `own_initializer_list&&` constructor.

```
class container {
    container( std::initializer_list< std::string > il )
        : container( il.begin(), il.end() ) {}
    container( std::own_initializer_list< std::string > && ilref ) {
        auto il = std::move( ilref );         // Move from the source object ASAP.
        this->assign( std::make_move_iterator( il.begin() ),
                      std::make_move_iterator( il.end() ) );
    }
    // …
```

4. Rationale

4.1. Naming

The previous revision of this proposal, N4166, used partial specialization to name the new class as `initializer_list<T&&>`. This has been changed because it tended to create the impression that `begin` and `end` would return `std::move_iterator`. The new name was chosen to reflect when and how it should be used, and how it is different. A list received by a function or constructor becomes its own. There are many alternatives:

- `mutable_initializer_list` (the name proposed in N2801): "Mutable" suggests an overridden `const`, and suggests usage as scratch space for manipulating values. Users should not be encouraged to do anything besides iterate over the list or pass it to a constructor.
- `movable_initializer_list`: "Movable" is a common term for rvalue semantic support, but this colloquial usage is at odds with the standard terminology. Anything that is copyable is also movable, though maybe not efficiently. The words also seem to be in a suboptimal order; this name seems like an approximation of `list_of_movable_initializers`.
- `unique_initializer_list`: The most essential feature of the new class is that aliasing is forbidden and object identity is guaranteed for sequence elements. Also, this name follows the pattern of `unique_ptr`. However, most users are unaware of the aliasing issue and care little about uniqueness of temporaries, making this name cryptic.
- `temporary_list` or `sequence_literal`: These are terse and descriptive of their own pure functionality, but do not express the relationship to the original `initializer_list`.

Losing partial specialization sacrifices generic functions of the form `template<class T> void foo(initializer_list<T>)`, which would accept an owning (`<T&&>`) or non-owning (`<T>`) list. However, it is better to write generic functions that accept any sequence type at all.

4.2. Deduction

N4166 avoided breakage by exactly preserving the rules for deducing `auto` from an initializer list. On further consideration, the use cases outweigh the risks. Renaming the class and avoiding partial specialization has also eliminated the danger of mistaking an owning list for a non-owning one. The only routes to undesirable behavioral change from this proposal are:

- Sequence elements change from `const` to cv-unqualified. Overloaded functions that detect `const` to opportunistically modify an argument may cause problems. However, such overloads are terrible practice. It is not uncommon for reference wrappers to do something similar, but one shouldn't retain references to something as transient as an `initializer_list` sequence.
- The name of an owning list cannot be used to initialize a variable or parameter declared as `auto`. An `auto &&`, `auto &`, or `auto const &` reference needs to be declared instead. This sanitizes code, particularly parameters, as `initializer_list` implements reference semantics.
- The type of a named owning list may be observed using `decltype` and passed to a metafunction that only expects `initializer_list`. This is an edge case.

Note that derived-to-base conversions are allowed during template argument deduction, so an `own_initializer_list<foo>` argument will successfully select and deduce `T` in a generic `initializer_list<T>` parameter:

```
template< typename elem >
int fun( std::initializer_list< elem > );

auto sl = { std::string( "hello" ) };                    // Owning list.
int q = fun( sl );                                       // OK: slicing.
```

This ensures compatibility of `own_initializer_list` with legacy interfaces.

4.3. Class derivation

`own_initializer_list` inherits publicly from `initializer_list` so that the derived-to-base conversion enables passing it to existing functions. This is also the simplest and most efficient implementation, with no need for additional temporary objects. “Slicing” by a user-defined conversion does not produce a workable implementation, because there is only one user-defined conversion slot in an implicit conversion sequence, and it is already occupied by any `initializer_list` constructor ([over.ics.list] §13.3.3.1.5/6).

The class `initializer_list<T>` intrinsically has the semantics of a `const&` reference, and `own_initializer_list` is like its non-const referent. Conversion of `own_initializer_-list` to `initializer_list` is thus analogous to binding a non-const lvalue or an xvalue to a `const&` reference. The derived-to-base conversion is the nearest step on the implicit conversion ladder to cv-qualification conversion, which is really the most appropriate. (It may also be noted that if users had been prohibited from declaring or modifying `initializer_list` objects in the first place, as with `std::type_info`, this proposal would be much shorter — we’d only have to define semantics for `initializer_list&&` and no new class would be needed.)

4.4. Ownership

Already in current implementations, an `initializer_list` object and its sequence have equal lifetimes unless the static storage optimization applies (or sequence lifetime extension fails to apply, which is usually an error). Fortunately, this optimization is inapplicable to classes with nontrivial destructors, including most with move semantics. The existence of move semantics is sufficient to surmise that the optimization is inapplicable and to remove the unnecessary `const` qualification from the sequence. The intent is that the user should only leverage the modifiable access to move from the sequence, so it is framed as ownership.

The sequence is destroyed by the `own_initializer_list` destructor — of the object currently owning the list — to free resources at the earliest opportunity. There are several advantages to this, over the status quo of a scoped array object:

- Moved-from values are not necessarily resource-free. This depends on the class’ allocation scheme, whether the sequence elements have the same type as the target objects at all, and whether move semantics have been implemented for all subobjects of the elements. Destructors free resources more reliably than move constructors or assignment operators.

- Unwinding (e.g., from a `bad_alloc` exception) should free resources as soon as possible, not when an arbitrary outer caller is reached. Note, in this case, some or all sequence elements may not be moved-from.
- Destroying the sequence is more like the behavior of `std::vector`, which is one of the better existing workarounds. Preserving the sequence is only reasonable from the perspective of the array object, but that is supposedly a hidden implementation detail.
- Eliminating moved-from values also removes the temptation to reuse them. A user who really wants to, may pass `std::move_iterator` to preserve the sequence as a scratch space.

Moved-from `own_initializer_list` objects do not provide access to the sequence. It is disallowed to pass ownership to a new list, then destroy it and continue with the old list. Shared apparent ownership and aliasing between lists are prevented.

Overloads such as standard container constructors, which accept and copy from an lvalue `initializer_list`, may preserve this functionality while adding rvalue semantics by adding an `own_initializer_list&&` overload. To apply the moved-from state to the parameter, such a function should move-construct a local variable from it before iterating over the sequence. List lvalues are seldom used, and libraries may choose not to support the functionality. A pass-by-value parameter overloaded over `initializer_list` and `own_initializer_list` will correctly and optimally handle all arguments that are *braced-init-lists* or rvalues.

In addition to the static storage optimization, there exists subtler a loop hoisting variant, which moves the sequence to an outer scope rather than to a global object. It is similarly only applicable when sequence construction and destruction lack side effects. The same principle of inspecting movability applies. Usually this optimization is applied to numeric types which are trivially movable. Under this optimization, the sequence only backs one unique list object at a time, so it is actually compatible with ownership, as long as the sequence contains correct values at the start of each loop iteration.

4.4.1. Usage restrictions

It is ill-formed to return, throw, or `new`-allocate an `own_initializer_list` or to have one as a subobject. These cases are already broken for `initializer_list`, and although it is technically possible to avoid using any dangling references, such technique is esoteric. The community has formed a strong consensus that "`initializer_list` is not a container."

The restrictions are intended to balance maintenance of the status quo against teachability and usability: To maintain the status quo, `own_initializer_list` is a drop-in replacement for valid use cases, including move-construction. It is forbidden by diagnosable rules in exactly the cases where the weakness of `initializer_list` must already be taught. It is allowed in function signatures, and only prohibited in return statements, to facilitate uniformity in type system. Prohibiting return types would cause the infection to spread to the type system as well.

4.4.2. Implementation model

The sequence is allocated on the stack, and hence deallocated at the end of some scope. However, if ownership is transferred, it is destroyed before it is deallocated. Users are encouraged to take `own_initializer_list` at face value. Accept that it offers the best of both worlds (the heap and the stack). Let the compiler worry about making it work.

Several models are viable to describe or implement the details, for example, an `optional` array object, or an `aligned_storage` buffer used with placement new. The prototype works by creating a bona fide `T[N]` array, but removing its destructor from the AST.

4.4.3. Predynamic storage

Nontrivial destructors may soon be compatible with constant expressions. An initializer list of literal, but nontrivially-movable objects could be used to initialize a `constexpr` container object. Move semantics would still be appropriate, and applicable within the constexpr evaluation context. Although such a sequence would be forbidden the static storage optimization, its entire lifetime would be contained within the evaluation of a constant expression, so it would not exist at runtime.

If ever this proposal does prevent allocation of a sequence in ROM, the optimization may be restored by explicitly specifying `std::initializer_list<T>` instead of relying on deduction.

4.5. Isn't one class enough?

The fundamental problem with `initializer_list` is that it implements `const&` reference semantics without mentioning `const` or references in its type. If it had a `const` somewhere, we could define the non-`const` version and be done, without adding a new class.

Replacing `initializer_list` with a reference type comes close to being a good solution:

```
template< typename elem >
using initializer_list = own_initializer_list< elem > const &;
```

This would break the `initializer_list::iterator` member type, and the reference wouldn't behave the same the class when used as a nonstatic member. Close, but no cigar.

The next best one-class alternative is to deprecate (or discourage) `initializer_list` in favor of `own_initializer_list const &`, which is functionally equivalent except that it provides `const_iterators`, not `iterators`. In this case, the name `own_initializer_list` seems lengthy and overspecific. Something like `sequence_literal` would be better. A little further refinement would be needed to enable the static storage optimization when binding an initializer list to a `sequence_literal const &` reference.

Mass migration away from `std::initializer_list` may be overkill, but perhaps users should be given `std::sequence_literal` and `std::sequence_literal const&` as an alternative, uniform, compatible, coexisting style. For example, `own_initializer_list` could be an alias template to `sequence_literal`.

5. Conclusion

`std::initializer_list` support for non-copyable types has been sorely missed. An extensive body of literature teaches various workarounds to the problem, which all involve bad techniques that defeat the ownership semantics at the foundation of C++. The proposed solution is the optimal compromise, surgically designed and prototyped within the original model.

Not only is `std::own_initializer_list` good for its own use cases, it improves upon its foundation. Treating `initializer_list` as a value type for a return statement or a class member is a common mistake, leading to frequently-asked questions. Although this proposal does not directly address the classic `initializer_list`, it officially establishes the limitations in a way that compilers must diagnose poor usage of `own_initializer_list`, and might as well extend the courtesy to `initializer_list` as well.

5.1. Future work

N4166 has been prototyped in GCC, with standard library support. The incompatible changes in this proposal, specifically, renaming the "`<T&&>`" specialization to "`own_`," and supporting `auto` deduction, remain to be done at this date.

Normative wording has yet to be forged for this proposal.

The static sequence storage optimization should not be particular to `initializer_list` elements. It should be allowed for all prvalues that do not bind to (mutable) rvalue references. In fact, the optimization is not conforming to the letter of [dcl.init.list] §8.5.4/6 except when the `initializer_list` already has static storage duration, because the user may observe the addresses of sequence elements. A proposal should comprehensively address the storage and uniqueness of constant objects. The problem scope even extends to linkage of constant globals.

5.2. Acknowledgements

Rodrigo Campos, the author of N2719 and N2801, provided helpful feedback and kind encouragement.

Ville Voutilainen, Andrzej Krzemieński, and Tomasz Kamiński provided valuable review.