Initializer lists and move semantics

Rodrigo Castro Campos, rcc.dark@gmail.com
2008-09-30.
N2801=08-0311.

0. Changes since N2719

- Changed the name of the proposed class `initializer_list<T&&>` to `mutable_initializer_list<T>`.
- Some changes to what is proposed. The general idea remains the same but the current solution is shorter, cleaner and easier to implement.
- Fixed some technical errors. In particular, changed the integration with the standard library - no more themself-templated `initializer_list<typename mutability_chooser<E>::type>` constructors, this is impossible to implement as-is due to well-known template deduction limitations with nested template types althought it could be easily solved if we considered the mere choosing between mutable and inmutable initializer lists a special case (templates are unnecessarily general in this case, thus causing the type deduction limitations); this had its benefits.
- Concepts are used in the integration with the standard library to avoid unnecessary template uses.
- Fixed some typos.
- Added more examples, clarifications and wording.

1. Motivation

The future standard (known as C++0x) will include features that will make us possible to write cleaner, more elegant and more efficent code than was possible in C++98. However, the integration of these features (between them and with the current language) is as important as the features themselves.

I present an example that shows a (serious) omission that puts in danger the feasibility to express simple programs as shown below.

```
class nocopy_yesmove;
class yescopy_yesmove;

template<class T>
class c {
private:
    std::array<T, 3> e_;    // or T e_[3] for the matter, array is unfortunately not better

public:
    c( )
    {
    }

    c(const c& p)
    : e_(p.e_)
    {
    }

    /*
        c(c&& p)
        : how do we move? -we can't-
        {
        }
    */
}


int main( )
{
    c<nocopy_yesmove> c1;
    c<nocopy_yesmove> c2 = c1;          // not copyable, error
    c<nocopy_yesmove> c3 = move(c1);    // moveable but we can’t move!, error

    c<yescopy_yesmove> c4;
    c<yescopy_yesmove> c5 = c4;         // copyable, performs a copy
    c<yescopy_yesmove> c6 = move(c4);   // moveable but we still must copy
}
```

In fact, using three different class members `e1_`, `e2_`, `e3_` instead of `T e_[3]` is actually better. In big or high performance systems this will preclude the use of array for almost all cases and raise the use of homecooked solutions that are flexible enough, leaving `std::array` as something "probably" more efficent than `std::vector`, but not as flexible even if we know the max size at compile time. I consider this embarassing.

2. Can we fix `T[N]`?

No. Unfortunately, the notion of a c-array is so deep in the language that I consider it impossible to fix.
We can't even pass a `T[N]` as a function parameter without losing both the facts that its an array and its size.
This would imply that even if we could fix the problem locally, we can't forward it to another function.
On the other hand, we don't necessarily need to fix `T[N]`.

3. Can we fix `std::array`?

Not without some (very!) clever hacks (uses of `tuple`). We would prefer something like this:

```
template<typename T, size_t N>
class array {
public:
    array( );                       // default constructor
    array(const array&);            // copy constructor
    array(array&&);                 // move constructor
    array(initializer_list<T>);     // initializer list constructor
    //...

private:
    typename aligned_storage<sizeof(T)>::type buffer_[N];       // no longer T[N]
};
```

While we can now copy and move an `std::array`, this is not optimal. There are two related problems, one of them serious, and both consequence of `std::array` no longer being an aggregate:

- n2215 proposes to allow direct initialization to aggregates, allowing not to require a copy constructor for initialization. This was not the case in C++98, which required it to be accesible (the fact that a compiler can bypass the copy in some cases as an optimization is not of interest right now). This would make a c-array superior for initialization in C++0x. This is not good, however is not that bad since there is no current legal code that can do that anyway. C++0x is just about to allow it.
- We can't no longer do the following:

```
    nocopy_yesmove a, b;
    cin >> a >> b;

    array<nocopy_yesmove, 2> c = { move(a), move(b) };       // initializer lists must copy!, error
```

While this works as intended for c-arrays, it is not a limitation to `array`, but to initializer lists in general when dealing with non-aggregates; the same will happen to `vector`:

```
nocopy_yesmove a, b;
cin >> a >> b;

vector<nocopy_yesmove, 2> c = { move(a), move(b) };       // initializer lists must copy!, error
```

4. The library-only solution that fixes most cases.

This solution requires no changes to the language and allows more uses of initializer lists. This is not my prefered one, but could work if its decided not to modify the language.

```
// current declaration of containers

template<typename T >
struct container {
    container(initializer_list<T>);
    //...
};

container<int> c1 = {1, 2, 3};          // ok
container<nocopy_int> c2 = {1, 2, 3};   // error inside constructor, can't copy
```

The second line will currently fail because it expands to

```
const nocopy_int _arr[] = {nocopy_int(1), nocopy_int(2), nocopy_int(3)};
container<nocopy_int> c2(initializer_list<nocopy_int>(_arr, 3));        // error, can't copy
```

The array is fully type-complaint before its really needed. `nocopy_int` comes in play too early, so we can't copy inside the sequence constructor.

However, it could compile if we change the declaration of the sequence constructor to:

```
template<typename T >
struct container {
    template<class E>
    container(initializer_list<E>);
    //...
};


container<int> c1 = {1, 2, 3};          // ok, initialized with initializer_list<int>
container<nocopy_int> c2 = {1, 2, 3};   // ok, also initialized with initializer_list<int>
```

The `Convertible` concept can come in handy to respect `explicit` constructors. Similar example:

```
container<string> c3 = {"a", "b"};           // initialized with initializer_list<const char*>
container<nocopy_string> c4 = {"a", "b"};   // also initialized with initializer_list<const char*>
```

Sequence constructors will be probably used with `constexpr` expressions and literals of primitive types 90% of the time, so we could say we have the 90% of the solution. However this will still fail:

```
nocopy_int nci;
container<nocopy_int> c2 = {move(nci), nocopy_int(2), nocopy_int(3)};   // error
```

5. The complete solution

Can we fix the last error? Yes, we can fix that too, but we need to add mutable initializer lists to the language.

```
template<class T>
struct mutable_initializer_list {
    //...

    T* begin( );                              // mutable interface to the underlying array
    T* end( );                                // mutable interface to the underlying array
    initializer_list<T> to_const_list( ) const;
};


void f(initializer_list<string>);             // takes an inmutable initializer list of strings
void g(mutable_initializer_list<string>);     // takes a mutable initializer list of strings
```

Requesting a mutable initializer list that mantains the expected semantics will force us to construct the initializer list, per use, in runtime:

```
void f(initializer_list<int>);
void g(mutable_initializer_list<int>);

void h( )
{
    f({1, 2, 3});      // this initializer list can be generated in compile time, can be placed
                        // in readonly memory and/or shared

    g({1, 2, 3});       // this initializer list must be generated in runtime and can't be placed
                        // in readonly memory and/or shared despite being initialized with constants
}

h( );
h( );
```

The initializer list used to call `f( )` needs only to be constructed once (most likely in compile time), can be placed in readonly memory and can shared between several calls to `h( )`.

On the other hand, the initializer list used to call `g( )` needs to be constructed per use in runtime (that means, constructed in each call to `h( )`). It cannot be shared and cannot be placed in readonly memory.

The rewrite rule for `h( )` is simple:

```
void h( )
{
    const int _arr0[] = {1, 2, 3};             // const array, optimizable given the chance
    f(initializer_list<int>(_arr0, 3));

    int _arr1[] = {1, 2, 3};                   // non-const array, can’t be optimized
    g(mutable_initializer_list<int>(_arr1, 3));
}
```

and the consequences are fully shown in the following example:

```
void g(mutable_initializer_list<int>);

for (int i = 0; i < 1000; ++i) {
    g({1, 2, 3});          // g( ) is requesting a mutable initializer list, the list must be
                           // constructed and destroyed in each iteration
}
```

We should remember that the programmer is making this request explicitly, there is no magic or hidden performance penalty involved. The lost optimizations are no worse than the ones lost when the initializer list’s contents are not known in advance. Mutable lists are also required to be *rvalues*, since they work as expected even with this restriction.

The good news are that, thanks to mutable initializer lists, we can do the following:

```
template<typename T>
struct reallocatable_sequence_list {
    typedef initializer_list<T> type;
};


template<std::MoveConstructible T> requires !std::CopyConstructible<T>  // in <concepts>
struct reallocatable_sequence_list<T> {
    typedef mutable_initializer_list<T> type;
};
```

```
template<class T>
struct container {
   container(typename reallocatable_sequence_list<T>::type in)
   // we need mutable lists sometimes, we use reallocatable_sequence_list to choose
   {
       T element_from_container = move(*in.begin( ));
       // move takes constness in count, works as expected
   }
};

container<int> c1 = {1, 2, 3};                // initializer_list<int> generated

nocopy_int nci;
container<nocopy_int> c2 = {move(nci), 2, 3};
                                              // mutable_initializer_list<nocopy_int> generated
```

Its easy to see how the type, in the presence of `reallocatable_sequence_list`, chooses the initializer list mutability.

Since in this case inmutables list are unusable when the types can't be copied, the lost optimizations are not to blame; its impossible to use initializer lists otherwise. The only performance overhead would be moving the elements and calling their destructors (move semantics are non-destructive). Under favorable circumstances (in regard to inlining and exceptions), the compiler could optimize their use.

Mutability is, obviously, not always needed even for lists of non-copyable types. Optimizations are still possible in these cases:

```
template<class T>
void print(initializer_list<T> in);
// no need to use reallocatable_sequence_list, this list is always inmutable

print({1, 2, 3});          // initializer_list<int> generated, optimizable
print({nocopy_int(1), nocopy_int(2), nocopy_int(3)});
                           // initializer_list<nocopy_int> generated, optimizable
```

Another example:

```
for (int i = 0; i < 1000; ++i) {
    m({some_value, f( )});
}
```

Given that `f( )` is `constexpr`, how many times is `f( )` needed to be called?

All of the following conditions must be true to force `f( )` to be called per iteration:

- `m( )` takes a `mutable_initializer_list<T>`.
- `f( )` returns some type `R` (is possible that `R` = `T`) and a `const R&` cannot be used to construct a `T`.
- `f( )` returns some type `R` (is possible that `R` = `T`) and an `R&&` can be used to construct a `T`.

`f( )` can be called only once in any other case.

These conditions are the ones the compiler will have to consider in real code while performing the optimizations. However, the fulfillment of the last two may vary from type to type. This could be considered bad for a programmer that wants an insane amount of control. However, I don't see any real problem with the implicit compiler-guided optimization since a `constexpr` function has no side-effects. The alternative code that would make obvious what is happening is:

```
for (int i = 0; i < 1000; ++i) {
    auto temp = f( );
    m({some_value, move(temp)});
}
```

which I don't really like. I propose it to be implicit, the programmer shouldn't really care in my opinion. If decided otherwise, we could require the compiler to generate an error and force the programmer to write the alternative version if `f( )` cannot be guaranteed to be called just once while mantaining the expected semantics. However, generating errors for missed optimizations is something I really dislike.

As we can see, even if `m( )` requested a mutable list `f( )` can still be called just once if `f( )` returns a copyable `T` or an `R` and `T` is constructible with a `const R&` (for example, `m( )` takes a `string` but `f( )` returns a `const char*`). The key in that case is making `f( )` return some kind of literal, which can be *rom*'ed and used to construct the type `m( )` expects.

6. What is needed from the language?.

- Being able to construct a `mutable_initializer_list<E>` with the braced list `{ a, b, … }` syntax.
- Giving `mutable_initializer_list<E>` the same treatment as `initializer_list<E>` regarding deduction of E, conversion to E (no narrowing), lifetime, *rvalue*-ness and preference in constructors.
- Disallowing sharing / read-only memory placement optimizations of `mutable_initializer_list<E>` as already explained.
- For `auto`-specifiers with braced lists and overloading between `initializer_list<E>` and `mutable_initializer_list<E>` (with the same E), giving preference to `initializer_list<E>`.

```
auto v1 = {1, 2, 3};              // decltype(v1) is initializer_list<int>
auto v2 = {nocopy_int(1), nocopy_int(2), nocopy_int(3)};
                                  // decltype(v2) is initializer_list<copy_int>

mutable_initializer_list<int> v3 = {1, 2, 3};
                                  // ok, mutable_initializer_list<T> needs to be explicitly required


template<class T>
void g(mutable_initializer_list<T>);

h({1.1, 1.2, 1.3});               // ok, mutable_initializer_list<double>


void g(initializer_list<char>);
void g(mutable_initializer_list<char>);
void g(mutable_initializer_list<double>);

g({'a', 'b', 'c'});               // g(initializer_list<char>) is called

mutable_initializer_list<char> v4 = {'a', 'b', 'c'};
g(v4);                            // g(mutable_initializer_list<char>) is called, explicit use of v4

g({1.0, 2.0});                    // g(mutable_initializer_list<double>) is called, best match
```

Overloading between `initializer_list<E>` and `mutable_initializer_list<E>` should be rare (I can't think of a use case), but simply giving preference to `initializer_list<E>` in this and for `auto`-specifiers cases is a good choice since mutable lists ideally should remain as an (ignorable) extension.

7. What is needed from the library?.

- Add the `reallocatable_sequence_list` template. Prohibiting user specialization is intended.
- Change the declaration of (most, an slightly different approach is needed for `std::array` since it is a fixed-size container) sequence constructors from

  ```
  template<class T>
  struct container {
      container(initializer_list<T>);
  };
  ```

  to

  ```
  template<class T>
  struct container {
      container(typename reallocatable_sequence_list<T>::type>);
  };
  ```

A separate paper is needed to adequate the library. Most changes are this simple, though.

8. Proposed wording.

In [8.5.4 [dcl.init.list]], paragraph 2:

Class templates std::initializer_list<E> and std::mutable_initializer_list<E> are called initializer-list types. A constructor is an initializer-list constructor if its first parameter is of ~~type std::initializer_list<E>~~ an initializer-list type or reference to a possibly cv-qualified ~~std::initializer_list<E> for~~ initializer-list type with some type E as template parameter, and either there are no other parameters or else all other parameters have default arguments (8.3.6). [Note: Initializer-list constructors are favored over other constructors in list-initialization ([over.match.list]).] The ~~template~~ templates std::initializer_list and std::mutable_initializer_list ~~is~~ are not predefined; if the header <initializer_list> is not included prior to a use of ~~std::initializer_list~~ any of them --even an implicit use in which the type is not named ([dcl.spec.auto])--the program is ill-formed. Objects of initializer-list types are called initializer-list objects.

List-initialization of an object or reference of type T is defined as follows.

1. If T is an aggregate, aggregate initialization is performed (8.5.1 [dcl.init.aggr]). [Example:

```
double ad[] = { 1, 2.0 }; // ok
int ai[] = { 1, 2.0 }; // error: narrowing
```

--- end example]

2. Otherwise, if T is a specialization of ~~std::initializer_list<E>~~ an initializer-list type, ~~an initializer_list~~ the corresponding object is constructed as described below and used to initialize the object according to the rules for initialization of an object from a class of the same type (8.5).

In [8.5.4 [dcl.init.list]], paragraph 4:

When an initializer list is implicitly converted to ~~a std::initializer_list<E>~~ an initializer-list type, the object passed is constructed as if the implementation allocated an array of N elements of type E, where N is the number of elements in the initializer list. Each element of that array is initialized with the corresponding element of the initializer list converted to E, and the ~~std::initializer_list<E>~~ corresponding object is constructed to refer to that array. If a narrowing conversion is required to convert the element to E, the program is ill-formed. [Example:

```
struct X {
  X(std::initializer_list<double> v);
};
X x{ 1,2,3 };
```

The initialization will be implemented in a way roughly equivalent to this:

```
double __a[3] = {double{1}, double{2}, double{3}};
X x(std::initializer_list<double>(__a, __a+3));
```

assuming that the implementation can construct ~~an initializer_list~~ a std:: initializer_list with a pair of pointers. The same applies to std::mutable_initializer_list --- end example]

The lifetime of the array is the same as that of the ~~initializer_list~~ initializer-list object. [Example:

```
typedef std::complex<double> cmplx;
std::vector<cmplx> v1 = { 1, 2, 3 };

void f()
{
  std::vector<cmplx> v2{ 1, 2, 3 };
  std::initializer_list<int> i3 = { 1, 2, 3 };
}
```

For v1 and v2, the ~~initializer_list~~ initializer-list object and array created for { 1, 2, 3 } have full-expression lifetime. For i3, the ~~initializer_list~~ initializer-list object and array have automatic lifetime. --- end example] [Note: For std::initializer_list, ~~The~~ the implementation is free to allocate the array in read-only memory if an explicit array with the same initializer could be so allocated. For std::mutable_initializer_list, no sharing or read-only memory placement can be applied. [Example:

```
void f(std::initializer_list<int>);
void g(std::mutable_initializer_list<int>);

for (int i = 0; i < 1000; ++i) {
    f({1, 2, 3});     // the underlying array can be shared and can be placed in read-only memory
    g({1, 2, 3});     // the underlying array must be constructed and destroyed in each iteration
}
```

--- end example] --- end note]

In [13.3.3.1.5 [over.ics.list]]:

When an argument is an initializer list (8.5.4), it is not an expression and special rules apply for converting it to a parameter type.

If the parameter type is ~~std::initializer_list<X>~~ an initializer-list type with some X as template parameter and all the elements of the initializer list can be implicitly converted to X, the implicit conversion sequence is the worst conversion necessary to convert an element of the list to X. This conversion can be a user-defined conversion even in the context of a call to an initializer-list constructor. [Example:

```
void f(std::initializer_list<int>);
f( {1,2,3} ); // OK: f(initializer_list<int>) identity conversion
f( {'a','b'} ); // OK: f(initializer_list<int>) integral promotion
f( {1.0} ); // error: narrowing

struct A {
  A(std::initializer_list<double>); // #1
  A(std::initializer_list<complex<double>>); // #2
  A(std::initializer_list<std::string>); // #3
};
A a{ 1.0,2.0 }; // OK, uses #1

void g(A);
g({ "foo", "bar" }); // OK, uses #3
```

—end example]

If after determining the best match of the candidates with an initializer-list type as the parameter type, an overload between std::initializer_list<X> and std::mutable_initializer_list<X> exists, std::initializer_list<X> is selected. [Example:

```
void m(std::initializer_list<int>);  // #1
void m(std::mutable_initializer_list<int>);  // #2
void m(std::mutable_initializer_list<double>);  // #3
m({1, 2});  // int is the best match, uses #1
m({'a', 'b'}); // int is the best match, uses #1
m({1.0, 2.0});  // double is the best match, uses #3
```

—end example]

In [14.8.2.1 [temp.deduct.type]], paragraph 1:

Template argument deduction is done by comparing each function template parameter type (call it P) with the type of the corresponding argument of the call (call it A) as described below. ~~If removing references and cv-qualifiers from P gives std::initializer_list<P0> for some P0 and the argument is an initializer list (8.5.4)~~ If the argument is an initializer-list type (8.5.4) and removing references and cv qualifiers from P gives the corresponding initializer-list object with some P0 as template parameter, then deduction is performed instead for each element of the initializer list, taking P0 as a function template parameter type and the initializer element as its argument. Otherwise, an initializer list argument causes the parameter to be considered a non-deduced context (14.8.2.5). [Example:

In [14.8.2.5 [temp.deduct.type]], paragraph 5 in final bullet at the top level:

A function parameter for which the associated argument is an initializer list ([dcl.init.list]) but the parameter does not have ~~std::initializer_list~~ an initializer-list type or reference to a possibly cv-qualified ~~std::initializer_list type~~ initializer-list type. [Example:

In [18.8 [support.initlist]]:

The header <initializer_list> defines ~~one~~ two ~~type~~ types.

Header <initializer_list> synopsis

```
namespace std {
  template<class E> class initializer_list {
  public:
    initializer_list();

    size_t size() const; // number of elements
    const E* begin() const; // first element
    const E* end() const; // one past the last element
  };

  template<class E> class mutable_initializer_list {
  public:
    mutable_initializer_list();

    size_t size() const; // number of elements
    E* begin(); // first element
    E* end(); // one past the last element
    initializer_list<E> to_const_list( ) const; // conversion to initializer_list<E>
  };
}
```

An object of type initializer_list<E> provides access to an array of objects of type const E while an object of type mutable_initializer_list<E> provides access to an array of objects of type E. [Note: A pair of pointers or a pointer plus a length would be obvious representations for ~~initializer_list~~ both types. ~~initializer_list is~~ These types are used to implement initializer lists as specified in 8.5.4. Copying ~~an initializer list~~ an object of any of these types does not copy the underlying elements. —end note]

18.8.1 ~~Initializer list~~ initializer_list constructors [support.initlist.cons]

initializer_list();
  Effects: constructs an empty initializer_list object.
  Postcondition: size() == 0
  Throws: nothing.

18.8.2 ~~Initializer list~~ initializer_list access [support.initlist.access]

const E* begin() const;
  Returns: a pointer to the beginning of the array. If size() == 0 the values of begin() and end() are unspecified but they shall be identical.
  Throws: nothing.

const E* end() const;
  Returns: begin() + size()
  Throws: nothing.

size_t size() const;
  Returns: the number of elements in the array.
  Throws: nothing.

18.8.3 mutable_initializer_list constructors [support.initlist.mutable.cons]

mutable_initializer_list();
  Effects: constructs an empty mutable_initializer_list object.
  Postcondition: size() == 0
  Throws: nothing.

18.8.4 mutable_initializer access [support.initlist.mutable.access]

E* begin();
  Returns: a pointer to the beginning of the array. If size() == 0 the values of begin() and end() are unspecified but they shall be identical.
  Throws: nothing.

E* end();
  Returns: begin() + size()
  Throws: nothing.

size_t size() const;
  Returns: the number of elements in the array.
  Throws: nothing.

initializer_list<E> to_const_list( ) const;
  Returns: an initializer_list<E> with access to the same array referenced by the mutable_initializer_list object.
  Throws: nothing.
  Requires: The program shall not treat the returned value as a valid initializer_list after any subsequent call to a non- const member function of mutable_initializer_list that designates the same object as this.

9. Acknowledges.

Thanks to Howard Hinnant and Bjarne Stroustrup for their comments.

10. References.

- Dos Reis, Gabriel., Stroustrup, Bjarne. (2007). Initializer lists (Rev. 3). : n2215.
- Stroustrup, Bjarne. (2008). Uniform initialization design choices. : n2477.
- Adamczyk, J., Dos Reis, Gabriel., Stroustrup, Bjarne. (2008). Initializer lists WP wording (Revision 2). : n2531.
- Stroustrup, Bjarne. (2008). Uniform initialization design choices (Revision 2). : n2532.
- Merrill, Jason., Vandevoorde, David. Initializer List proposed wording. : n2672.