Initializer lists and move semantics

Rodrigo Castro Campos, rcc.dark@gmail.com
2008-08-20.
N2719=08-0229.

1. Fast facts

- Initializer lists in sequence constructors (as proposed in n2215) have an inmutable underlying array.
- Inmutability allows optimizations (*ROMability*) that would be impossible to achieve otherwise in some important use cases. This is the primary reason that led to choose inmutability rather than mutability in initializer lists.
- However, inmutable initializer lists cannot be used as intended when copy semantics aren't available since the elements must be copied.
- An almost perfect solution (a perfect solution for almost all cases) can be built without changing the language.
- When move semantics are available, mutable initializer lists could be useful to solve the missing cases.
- The accepted proposal discussed between inmutability and mutability and chose the best, but never considered having both.
- This paper proposes having both types of initializer lists to reach the perfect solution (requiring language change).
- The almost perfect solution (requiring no language change) is quite achieveable and probably would be good enough considering current C++0x time constraints.

2. 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 cant 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.

3. 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]`.

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

Not without some (very!) clever hacks (for example, initializing `array` with `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 value_type[N] buffer_;       // 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 copy constructors 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
```

5. The almost perfect solution

The almost-perfect solution requires no changes to the language.

```
// current definition of containers

template<typename T >
class container {
public:
    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 definition of the sequence constructor to:

```
template<typename T >
class container {
public:
    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 elements from the container will be inplace-initialized with literals. 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
```

6. The perfect solution

Can we fix the last error? Yes, we can fix that too, but we need to add mutable initializer lists to the language. I propose a partial specialization that has a mutable interface to the underlying array:

```
template<class T>
class initializer_list<T&&> {       // partial specialization
public:
    //...

    T* begin( ) const;        // mutable interface to the underlying array
    T* end( ) const;          // mutable interface to the underlying array
};
```

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

Obviously, requesting a mutable initializer list will force us to construct the initializer list in runtime:

```
void f(initializer_list<int>);         // f( ) is requesting an inmutable initializer list
void g(initializer_list<int&&>);       // g( ) is requesting a mutable initializer list

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(initializer_list<int&&>(_arr1, 3));
}
```

and the consequences are fully shown in the following example:

```
void g(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 isn’t `constexpr`-initialized. Mutable lists are also required to be rvalues, since they work as expected even with this restriction.

Overloading of `initializer_list<T>` and `initializer_list<T&&>` is forbidden. It could be solved but not without introducing confusion and implementation-defined behavior.

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

```
template<class T>
struct sequence_init_type {
   typedef T type;                            // type is T by default
};

struct nocopy_int;
```

```
template<>
struct sequence_init_type<nocopy_int> {       // specialization: this type must be moved
   typedef nocopy_int&& type;                 // type is T&& in this case
};


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

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

nocopy_int nci;
container c2<nocopy_int> = {move(nci), nocopy_int(2), nocopy_int(3)};
                                       // mutable list initializer_list<nocopy_int&&> generated
```

Its easy to see how the need for using an already-existant variable instead of literals changes the way the template is instantiated. Since the list is no longer `constexpr` initialized, optimizations are inhibited anyway – mutable lists are not the ones to blame. We use literals when we can, the true type when we cannot.

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(std::initializer_list<T> in);
// no need to use sequence_init_type to switch to mutable lists, this is always inmutable

print({1, 2, 3});          // inmutable list initializer_list<int> generated, optimizable
print({nocopy_int(1), nocopy_int(2), nocopy_int(3)});
                           // inmutable list 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 an `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; exactly the same trick we were using for sequence constructors.

The last and most complicated example I can think of is `map` (or similar uses of `pair` and `tuple`):

```
template<class K, class V>
struct map {
    template<class E1, E2>
    map(std::initializer_list<typename sequence_init_type<pair<E1, E2>>::type>);
};


map<int, nocopy_int> m1 = {{1, 2}, {3, 4}};   // ok, initializer_list<pair<int, int>>

nocopy_int i1, i2;
map<int, nocopy_int> m2 = {{1, move(i1)}, {2, move(i2)}};
       // error if the specialization of sequence_init_type for pair<int, nocopy_int>
       // doesn't exist (without it, initializer_list<pair<int, nocopy_int>> would be
       // generated by default
```

The obvious solution would be to define a partial specialization of `sequence_init_type` for `pair` and `tuple` so mutable lists are generated for them if any of the types contained requires it. This would involve some easy template magic. Then we could write:

```
map<int, nocopy_int> m2 = {{1, move(i1)}, {2, move(i2)}};
       // nocopy_int requires mutable lists
       // ok, initializer_list<pair<int, nocopy_int>&&> is generated then
```

7. Acknowledges.

Thanks to Howard Hinnant and Bjarne Stroustrup for their comments.

8. 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.