N3201=10-0191
10/23/2010
Bjarne Stroustrup
Email: bs@cs.tamu.edu

Moving right along

Abstract

This note follows up on N3174 to further explore the relation between move and copy. It re-examines the conjecture that the issue of implicit generation of move operations is orthogonal to the issue of defining the state of a moved-from object. It refines my suggested solution to the problems with generated move operations by taking default construction into account. It discusses implementation issues and performance implications related to generated move operations. Finally, it presents a set of alternatives in my order of preference.

Move and Copy

In my note N3174 about generation of move operations, I say:

- copy and move are often invoked by the same notation
- move is often seen as an optimization of copy (some, but not all moves are just that)
- several of the problems we have encountered with move (in the language, library, and user code) have had a failure to consider copy and move together as its root cause

That's correct as far as it goes, but not precise.

For **x=y** move a copy both leaves **x** with the value that **y** had (or still has). After **x=y**, copy has **x==y** (or ought to have) and move leaves **y** in an "unspecified but valid" state. Thus, copy and move differ only in the state of the copied-from and moved-from objects.

Consider four kinds of types:

- Both copy and move makes sense (and move is an optimization of "copy followed by destroy")
- Move-only objects (where copy doesn't make sense)
- Copy-only objects (where move doesn't make sense)
- Object for which neither move nor copy make sense

The first two are the ones that provide the motivation for introducing move. Our problem is to – either by manually defining operations or through language rules – determine into which category a particular type falls. I'm deliberately non-specific about the meaning of "copy followed by destroy" and "doesn't make sense" to allow for people's varying notions of that.

Consider an example. A two-dimensional matrix could be defined as

```
template<type T, int d1, int d2> struct M2 {
        T elem[d1][d2];
        constexpr int dim1() const { return d1; }
        constexpr int dim2() const { return d2; }
        // …
};
```

This leaves little room for optimization of copy because every **M2** must have **d1*d2** valid elements of type **T**. Basically, it would be unreasonable to return a **M2** with even moderately large values of **d1*d2*sizeof(T)** by value. The obvious addition operator would be something like

```
M2& operator+(const M2&, M2&);      // allocate result on free store
```

(and somehow deal with memory management) or

```
void operator+(const M2&, const M2&, M2& res);      // caller allocates space for result
```

Neither is acceptable for plus (IMO), but neither technique for passing large results is uncommon.

Instead, we can define something like

```
template<type T> class Matrix2 {
        T* elem;
        int d1, d2;
public:
        Matrix2(int dd1, int dd2)   // simplified
                : elem(dd1*dd2>0?new T[dd1*dd2]:nullptr),
                  d1(dd1), d2(dd2)
                {}
        ~Matrix2() { delete[] elem; }
        // …
};
```

We can define a move that sets a **Matrix2** object to **{nullptr,0,0}** and cheaply and safely return a **Matrix2** from a function. For example, the obvious addition operator would be

```
Matrix2 operator+(const Matrix2&, const Matrix2&);    // move the result out of the function
```

This is the idiom I think we should promote: It is simple and efficient.

Moved-from state

I think that the original idea was that a moved-from object would never be used again. The canonical example was "**return x;**" where move would save us from an expensive copy of **x** followed by the destruction of **x**. It follows that the moved-from state must be destructible. It was soon decided that in

addition, it would help programmers (especially library builders) if it was possible to assign to a moved-from object. However, even that was not obvious to all and attempts to require further guarantees for the moved-from state elicited strong opposition. The consensus seems to be that a moved-from state must be "valid but unspecified" exactly as an object after a throw under the basic guarantee, but that nothing more can be guaranteed (except possibly for some specific type).

The obvious state of a moved-from object is the "empty state." The snag is that for some types there is no obvious "empty state" (e.g., consider proxies). How would we know if there was an empty state for a type **X**? Consider default construction. If there is a default constructor, we have an empty state – and by implication – we have an excellent candidate for the moved-from state. Assuming that we equate "establishing the invariant" with "constructing", the existence of a default constructor ensures that there is a way to establish the invariant without running multiple operations or supplying arguments. Ignoring details, the obvious default move becomes:

1. Move each element
2. Default construct the moved-from object

This observation leads to the idea: *Generate a move operation for **X** only if we can default construct an **X***. There may be other constraints on generation; in particular, the ones I suggested in N3174. Let's explore.

One way of looking at this suggestion is as a way of dealing with the cases where the memberwise moves breaks the invariant and/or doesn't leave the object in a "valid but unspecified" state.

Case 1: No default constructor

If default construction is disallowed for an **X** (i.e. there is a constructor, but no default destructor, or the default constructor has been **delete**d), it is unlikely that there is an obvious state to be used for a moved-from object. Maybe we better not generate a move operation: By implication the programmer has stated that there is no acceptable default state. If the programmer has a better idea, he or she can explicitly define a move constructor.

If **Matrix2** above has no default constructor, it is an example of this.

Case 2: User-defined default constructor

Use the default constructor to set the state of the moved-from object. That would take care of the **SomeonesClass** that started this round of discussions about move generation as well as **X** and the first **Y** from Dave Abraham's blog entry (N3153=10-0143). Using **X()** in addition to member moves would typically be more expensive than a simple series of member moves, but at least we know that it's a valid state. If the programmer knows an optimization, he or she can program it explicitly.

If **Matrix2** has

```
Matrix2::Matrix2() : elem(nullptr), d1(0), d2(0)  {};
```

it is an example of this.

Case 3: Generated default constructor

Use the generated default constructor exactly as a user-defined one. After all, if it was good enough for default objects, it's probably good enough as the moved-from state. Indeed, that logic is sufficient to handle moves for Dave Abraham's last two **Y**s.

If **M2** above had no constructors, it would get a default constructor and (under the FCD rules and my suggested rules) get move operations. However, these would be harmless and possibly even helpful in optimizing moves of element values. If a variant of **M2** stored its dimensions without using a constructor to do so, it might become a problem case (see below).

Generated default constructor problems

This simple scheme based on the default constructor correctly handles most cases. What kind of cases doesn't it handle?

A generated default constructor does not initialize a built-in type. For example, consider my first example in N3174:

```
class Vec {
        vector<int> v;
        int my_int;       // the index of my favorite int stored in v
        // implicit/generated  copy,  move, and destructor
        // …
};
```

We can default construct and get an object **{{},**uninitialized int**}**. The generated default constructor is not:

```
Vec::Vec() : v{}, my_int{} { }
```

Explicitly using **{}**, e.g., **Vec()** or **Vec{}** also gives us that, but default construction leaves **my_int** uninitialized (12.1 and 12.6.2). It is well known that this lack of initialization is an irregularity relative to the rest of the language and can cause problems. However, this is not the time to try to correct that.

An alternative solution would use **Vec{}** as the default value (giving **{{},0}**), but doing so has performance implications and it is not certain that it would be preferable in general. In particular, it is not obvious that setting **my_int** to **0** for the empty **v** makes more sense than leaving it unspecified.

The net effect is that if we (e.g., for **Vec**) establish an invariant without defining any constructor (e.g. through aggregate initialization or using an "**init()** function"), the result is a class for which the default copy may maintain the invariant but a generated move does not – even a generated move that uses the (generated) default constructor.

Consider my second example in N3174:

```
class Vec2  {
        vector<int> v;
        int* my_int;      // points to my favorite int stored in v
```

```
        // implicit/generated  copy,  move, and destructor
        // …
};
```

Both (generated) copy and move still breaks the (implicit) invariant.

Personally, I consider these two examples rare enough and strange enough to be worth breaking to avoid the problems related to a lack of default move described in N3174's section on "defaulting notation." What do I mean by "strange?" Nowhere in **Vec** or **Vec2** did I say what the value of **my_int** was supposed to be for an empty **v**. Thus, I'm not sure **Vec** and **Vec2** can be said to have meaningful or well-specified invariants.

If it is considered desirable to cope with such examples implicitly, **Vec** can be handled by using a default state defined by a default constructor that initializes all objects to their default value (as the **Vec::Vec()** above) rather than leaving built-in types undefined. **Vec2** is already "broken" in C++98 and beyond repair. I do not consider the added effort worthwhile.

Note that Dave Abraham's more realistic second and third **Y** examples do not suffer from the problem that **Vec** does because their **length** members are of a user-defined type with a default constructor, so they have a well-defined default state (as intended). **Vec** and **Vec2** were deliberately created to be worst-case scenarios.

Consider again **M2**. What if it wanted to store its dimensions? If it does so with a constructor (that is not a default constructor) move generation is (under my suggestion) disabled. If it does so with a constructor and provides a default constructor there is again no problem. Only if it uses some sort of **init()** function to set member variables (not constants) do we have a problem.

Performance implications

Are there performance implications from defining the moved-from state to the result of default construction? Yes, invoking the default constructor after running memberwise moves are going to be no cheaper than just running the memberwise moves.

The issue of whether a move of a built-in type should leave the source object unchanged or set it to **{}** was discussed in the EWG and the consensus was to prefer performance (no extra assignment) to the uncertain benefits of re-setting the source object. In the absence of a strong logical reason to do otherwise, we preferred performance.

Suggestion

So, I'm refining my suggestion of how to resolve the problem with generated copy and move operations: possible solution based on that:

1. Move and copy are generated by default (if an only if their elements move or copy as currently specified in the FCD)

   a. If any move, copy, or destructor is explicitly specified (declared, defined, **=default**, or **=delete**) by the user, no copy is generated by default.
   b. If any move, copy, or destructor is explicitly specified (declared, defined, **=default**, or **=delete**) by the user or if the class is not default constructible, no move is generated by default.
2. The state of a moved-from object of a class with a generated move operation is the default value for the class.

This suggestion imposes one more restriction on generation of move operations compared to the suggestion from N3174 which itself is strictly a restriction on what is currently in the FCD (e.g., 12.8[8]). In addition it suggests using default construction to defined the moved-from state for generated move operations.

As suggested in N3174, for C++0x we should only deprecate rather than ban implicit generation of copies in those cases.

Note that the moved-from state for a class with a user-defined move operation is still "unspecified but valid" because one use of user-defined moves is exactly to deal with classes and use cases for which the default state is not meaningful or expensive to establish.

As stated in N3174, I consider the implicit generation of copy and move operations important for ease of use. In particular, see the "Defaulting notation" section (pp5-7). In addition, I suspect that a heuristic (as suggested here) is as likely to be as correct (vis a vis implicit invariants and optimization opportunities) as an average programmer manipulating a set of controls.

It has been observed that it is impossible to craft a perfect set of rules and restrictions for when and how to generate move operations. I consider that obvious: for every language rule of any worthwhile expressive power, I can construct a misuse. It has also been observed (by Howard Hinnant, I think) that it would be unfortunate if the common cases of move were seriously complicated by "rare and arcane cases."

Implementation concerns

Several types of objections have been raised to the idea of defining the moved-from state of a generated move operation as the default state:

1. It's less efficient than just moving the elements
2. It violates language rules (e.g. by running a default constructor on an object that will be destroyed)
3. It simply can't be done that way (e.g. because of throwing default constructors)

My simplified answers to these simplified objections are:

1. not much for the cases where we really care about performance
2. no it does not because we are talking about an operation generated by the compiler

3. yes it can

The sections below go into some detail.

Before doing so, I'll point out that

- If we cannot generate"implicit moves" (at all or just not well), we have exactly the same problem with move operations (explicitly) defined **=default**.
- If move operations are not generated implicitly for types where move appears to make sense, application programmers will use **=default**. If they can't use **=default**, they will hand-code move operations and get them wrong exactly as they now do with copy operations (or worse).
- We are, of course, only talking about generated move operations. Programmers who carefully implement move operations by hand can do so as elegantly and efficiently as they know how.

I do not consider pushing the problem of defining move operations onto the application programmers a good solution. Doing so will lead to verbosity, errors, and/or underuse of move. The effect will be to seriously weaken the utility of move and most likely make the definition and use of move operations the domain of expert library builders.

Generated moves wouldn't be legal

One suggestion for implementing a generated move constructor is to move the elements and then default construct on top of the moved-from object. That obviously implies that the moved-from object is constructed twice and destroyed only once. However, the second construction is an implementation detail of an operation generated according to language rules: it is legal per definition. After moving the elements from the moved-from object, no invariant holds, and we can put the object into any "valid but unspecified" state we like in whichever way we please. In fact, even though I think that calling the default constructor will be the most common implementation technique, each implementation (compiler plus library facilities) should be free to generate that default state in the best way it can for each type.

In particular, calling the default constructor on the argument after moving the members would not lead to resource leaks because any resource represented by those members would have been transferred before the constructor was called on the arguments.

Note that the presence of a user-defined destructor inhibits the implicit generation of moves so moves are unlikely to be implicitly generated for resource handles.

This concern about destructors wouldn't even arise if a swap was used to replace the state.

For copy assignment, a destroy followed by a move construction would do the trick.

Generated moves can't be implemented

What about throwing move operations? In the EWG we initially wanted to ban them, but were persuaded that this was unrealistic. Maybe this decision was wrong and we should consider making

move operations **noexcept** by default, but for now, I'll assume that a move operation may throw. Similarly, I will assume that a default constructor may throw.

It seems obvious that to implement a move preserving one of the usual guarantees, we need a "primitive" copy or swap operation; that is an operation that copies bits without the possibility of a throw or a redefinition. A compiler can do that and so can a programmer using a casts, **memcpy()**, or something similarly low level, simple, and efficient.

How do we generate a move constructor for a class with a member move constructor that may throw? Construct a "temporary object" by moving into it. Once that's done, destroy the target object (for assignments only) and bit-blast (somehow) the temporary object into the target object. I deliberately refrain from using words such as "copy" and "move" here because casts may be needed to avoid undesirable semantic implications. We do not destroy the "temporary object"; it is not a real object. Rather, it is an implementation detail. Finally, we default construct the moved-from object.

How do we generate a move constructor for a class with a default constructor that may throw? We can again use a "temporary object." We default construct a "temporary object" and then bit-blasts it into the moved-from object. We do not need to destroy the moved-from object before changing its value, because we can assume that a moved-from object does not own resources.

Please note that the technique presented here in not required. Any equivalent technique may be used (e.g. swap based techniques). In particular, if we can assume that no exceptions can be thrown, the intermediary step using a "temporary object" can be eliminated.

Generated moves would be inefficient

Most move operations will not throw. Most default constructors will not throw. If we encourage the use of **noexcept** for move operations and default constructors, most classes outside library utility classes would not have to deal with exceptions.

We could implicitly declare a generated move operation **noexcept** (like destructors, assuming we go that way) unless it has a member with a (necessarily user-defined) potentially throwing move operation or default constructor. This would leave the (expensive) handling of the possibility to

(1) generated moves of types with user-defined moves that has not been declared **noexcept**
(2) user-defined moves

A more radical solution would be to suppress the generation of move if a class has a member with a move that is not **noexcept**.

Obviously, the suggestion made above (define the moved-from state as the default state for generated move operations) is potentially more costly than the original (and FCD) definition of "just move the members." However, it is not obviously much more expensive for the simple types for which performance is important. Consider Howard Hinnant's example in the move discussion:

class gslice

```
{
  valarray<size_t> __size_;
  valarray<size_t> __stride_;
  valarray<size_t> __1d_;

public:
  gslice()                              = default;
  ~gslice()                             = default;
  gslice(const gslice&)                 = default;
  gslice(gslice&&)                      = default;
  gslice& operator=(const gslice&)      = default;
  gslice& operator=(gslice&&)           = default;
  // ...
};
```

A “simple move” would look like this:

```
gslice::gslice(gslice&& a)
        : __size _(a.__size _),
        __stride_(a.__stride_),
        __1d_(a.__1d_)
{
}
```

A move that sets the state to the default is:

```
gslice::gslice(gslice&& a)
        : __size _(a.__size _),
        __stride_(a.__stride_),
        __1d_(a.__1d_)
{
        a.__size _ = valarray<size_t> ();
        a.__stride_ = valarray<size_t> ();
        a.__1d_ = valarray<size_t> ();
}
```

This seems to imply a something like a doubling of the number of memory accesses. There seems to be a simple choice between speed (the simple solution) and maintaining invariants that are established by constructors, but maybe not by simple moves of all members.

Conclusions

Here are the alternatives for generation of move operations that I have considered in order of preference:

1. Generate move operations as described in the "suggestion" section above (briefly: generate unless a user-specified copy, move, or destructor is declared (e.g., **=default**), using the default state as the moved-from state)
2. Generate move operations as described in N3174 (briefly: generate unless a copy, move, or destructor is declared (e.g., **=default**), using the state resulting from member moves as the moved-from state). This breaks more invariants than [1] but is simpler to implement.
3. Generate move operations unless a copy operation is declared (e.g., **=default**). This is the FCD status quo which will become the standard unless we see a large majority for an alternative)
4. Generate move operations only if the programmer asks for it using **=default**.
5. Never generate move operations.

I am of course always open to arguments, but my current position is that I can support [1], [2], or [3] only.

Note added 11/11/10: After the Core Working Group discussion, I agree that alternative [2] is preferable to alternative [1] because of its greater simplicity.

Acknowledgments

Achilleas Margaritis has been thinking along the same lines as is presented here and suggests that move can be defined by a swap with the default state: http://thegreatlambda.blogspot.com/2010/10/c-move-semantics-alternative-that.html .