N3174=10-0164
10/17/2010
Bjarne Stroustrup
Email: bs@cs.tamu.edu

To move or not to move

Abstract

This note presents a few troublesome examples related to implicitly generated copy and move operations and tries to place them into a common framework. I conclude that completely avoiding implicit generation of move operations will not solve the problems related to implicit invariants that have been discussed. I make a suggestion that might lead to better error detection and less implicit breakage. However, my main purpose is to collect issues and concepts so that we are less likely to do unintentional damage by a localized fix.

The suggestion is to generate move operations for a class if and only if there is no programmer-specified move, copy, or destructor. Similarly, to maintain a coherent set of rules, generating copy operations should be deprecated for classes with a programmer-specified move, copy, or destructor.

Should move operations be generated by default?

The reasons to prefer move operations to be generated by default are

- The most common case should be the easiest to write
- That most efficient case should be the easiest to write
- The rules for copy and move should be very similar

Conversely, the reasons for not generating are

- Generated move can break some old code
- Generated move can lead to surprises in new code

This note tries to evaluate the pros and cons. In particular, it considers the likely impact on users.

We have a set of reasonable, but mutually incompatible ideals:

- No breakage of C++98 code
- No verbosity in C++0x compared to C++98
- Very similar rules for copy and move
- No new “traps and pitfalls”
- A simple set of rules (sufficiently simple for real-world programmers to use)
- No major changes to the FCD

I consider this a language design problem, rather than a simple backwards compatibility problem. It is easy to avoid breaking old code (e.g. just remove move operations from C++0x), but I see making C++0x a better language by making move operations pervasive a major goal for which it may be worth breaking some C++98 code.

I try to be specific about which rules I assume for each example, but my default is what the FCD offers: default implicit generation of move and copy operations. Since we have an FCD this rule should only be changed after serious consideration and with a broad consensus.

Implicit move breaks implicit invariants

Consider a simple type

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

I could have expressed this more tersely as **pair<vector<int>,int>**. It is an example of the kind of type that caused a vigorous (and occasionally unrestrained) series of reflector messages. **Vec** has what I call an *implicit invariant*: There is a relation between two objects (here, the members) that is nowhere stated in declaratively in the code.

The implicit nature of invariants in C++ may be at the root of our problems with copy and move. We have no specific way of saying what invariant a class might be (trying to) maintain. In the absence of such a specific way of stating invariants, we have to look to language uses as indicators.

For **Vec**, the implicit (generated) copy maintains the invariant whereas the implicit (generated) move does not. For example, if we have

```
Vec x = { {1,2,3,4},2 };
Vec y = x;
Vec z = move(x);
```

Here, copy preserves the invariant (**y** is **{{1,2,3,4} 2}** just like **x**). However, move does not maintain the invariant as it moves from **x**: the value of **x** has become **{{},2}** and **2** isn't a valid index into **{}**.

You might point out that "nobody should access a moved from object, so this isn't a real problem." However, things are not that simple when a library (e.g. the standard library) might move from objects passed to it.

Implicit (generated) copy suffers from a similar problem

```
class Vec2  {
```

```
        vector<int> v;
        int* my_int;     // points to my favorite int stored in v
        // implicit/generated  copy,  move, and destructor
        // …
};
```

Obviously, after a default copy, **my_int** will in both copies point to an element in the source object. We may deem this "bad code that deserves to be broken" or "unrealistic", but this example demonstrates that the problem with a generated move has an exact counterpart for copy (which we have lived with for 27 years). Note that **Vec2** is roughly equivalent to **pair<vector<int>,int*>**, so that any problem we might have with **Vec2** can also appear for **pair** (and any other aggregation of objects).

A more realistic problem with implicit (generated) copy is the classic:

```
class Vec3  {
        int* elem;;
        int sz;   // elem points to points to sz elements
        // implicit/generated copy and move
        // …
        ~Vec3() { delete[] elem; }
};

void f()
{
        Vec3 v1(100);
        Vec3 v2 = v1;    // copies v1.elem
}  // here we delete the same elements twice
```

A move operation takes a non-const argument and typically mutates its source ("turns it into a Zombie") whereas a copy operation (typically) takes a const argument and does not mutate its source. However, this is not as fundamental a difference as it first appears because a copy can still render the source unusable (can be considered to have violated the source object's invariant). Consider:

```
const Vec3 v(100);

void f()
{
        Vec3 vv = v;     // copy
}  // here we delete v's elem
```

After a call of **f()**, any use of **v** is suspect (to put it mildly).

I note that this "breakage of invariant" was possible only because I defined a destructor (but not a copy constructor). Thus, the invariant for **Vec3** ("the object owns the elements") was partially specified but we failed to completely specify it.

I think we must approach the problem from a perspective of "pure language design" and then separately consider what we can actually do in real-world C++.

Implicit and explicit invariants

So, some implicit invariants are broken by implicit (generated) moves and copies. In 1984, I missed the chance to protect us against copy and we have lived with the problems ever since. I should have instituted some rule along the lines "if a class has a destructor, no copy operations are generated" or "if a class has a pointer member, no copy operations are generated." The former rule would consider the declaration of a destructor evidence of a non-trivial invariant (thus making the invariant explicit). The latter rule would be a pure heuristic. In N2904==09-0094 (the paper that proposed generation of moves), I tried again (for the Nth time) to introduce a rule to suppress generation of copy operations and (again) failed because we agreed (me included) that there was too much code out there that would be broken.

In the abstract, a good language design would suppress copy and move in a class with a user specified destructor (or some similar explicit "statement" from the programmer). For a new language with a suitable terse notation for requesting generation of default operations, this would lead to the fewest surprises and (on average) the earliest detection of mistakes.

Would it be better to *never* generate moves and copies? I doubt it. Copy and move are such fundamental and common operations that we would simply force everybody to say (somehow) "you can copy/move objects of my class." Think call-by-value, **vector** elements (e.g. **resize()** and **push_back()**), **pair**, **tuple**, function objects for algorithm policies, **swap**. Maybe a user doesn't copy much (e.g., people using an OO style of programming tend to avoid copying, preferring systematic use of pointers and references), but our STL –style library interfaces and implementation techniques are heavy with copies (that we'd like to optimize using moves).

Please note that "no generation" will not lead to "no surprises or bugs." Programmers would soon discover that the default (no copies of moves) was not the most common case and get into the habit of thoughtlessly requesting the most common useful case (e.g., **{cm}** for "generate copy and move"). In the absence of a concise notation for enabling copies and moves, some will resort to macros.

So,

1. Always generating moves and copies breaks some implicit invariants
2. Never generating moves and copies impose a burden on every programmer
3. Partially explicit invariants (e.g. explicit constructor but implicit copy) can cause serious errors given either implicitly generated copy or move.

My impression is that everyone who has looked at this has come to the question

- "When and how do we suppress generation of copy and move?" (given generation by default)

or its inverse

- "When and how do we request generation of copy and move?" (given no generation by default).

Defaulting notation

In an ideal world, I think we would decide on "no generation" as the default and provide a really simple notation for "give me all the usual operations." The reason is that the really subtle implicit invariants, e.g. **Vec** and **Vec2**, are in general undetectable by heuristics and therefore any "generate copies and moves by default" policy would lead to some surprises. Also, a "no default operations" policy leads to compile time errors (which we should have an easy way to fix), whereas a generate operations by default policy leads to problems that cannot be detected until run time.

As an aside, I consider partial and implicit invariants examples of brittle, error-prone code that ought to be discouraged.

Note that below, I will argue that in the absence of a really simple notation for defaulting copy and move operation, the "no implicit generation" approach will fail to deliver the safety it was meant to provide.

Assuming no copy or move by default, what would the user have to do? Consider

```
struct Rec {
        string name;
        char* addr;
        int id_number;
};

vector<Rec> v;
v.push_back({"M. Mouse", "Orlando", 666});    // error: cannot copy Rec
```

We have to modify **Rec**:

```
struct Rec {
        string name;
        char* addr;
        int id_number;

        Rec(const Rec&) = default;
        Rec(Rec&&) = default;
        Rec& operator=(const Rec&) = default;
        Rec& operator=(Rec&&) = default;
};

vector<Rec> v;
v.push_back({"M. Mouse", "Orlando", 666});    // error: cannot copy Rec
```

That trebled the source code size for **Rec** (194 character vs. 64 characters). Obviously, this short example used only a single copy (or move) constructor, so we needn't have added all of

```
Rec(const Rec&) = default;
Rec(Rec&&) = default;
Rec& operator=(const Rec&) = default;
Rec& operator=(Rec&&) = default;
```

However, adding only one of these operations would simply lead to later errors when **Rec** was used in different ways.

Even in an ideal world, this would give us food for thought. And we don't live in an ideal world. We have had implicit copy since the earliest days of C. A few years ago, I was arguing for a very terse form of default control; for example, **{=}** for "enable default copy." However, that was (possibly correctly) shot down for being obscure. However, I suspect given that copy is by default allowed (and has always been), we have to start with the assumption that default copy is allowed.

A more serious problem with defaults of all sorts is that the current rules are suitably conditional. They provide a copy iff all elements have copy and provide a move iff all elements have move. That works nicely for a large number of cases, including the motivating cases for the introduction of generated moves (e.g., **pair** and **tuple)**. If people have to use explicit defaults either the **=default** must be conditional (on element operations) or programmers will have to do elaborate programming based on the properties of elements (lots of **enable_if**, I guess). For example:

```
template<class X, class Y> struct My_pair {
        X first;
        Y second;
};
```

Under the current (FCD) rules, I can copy a **My_pair** iff **X** and **Y** can be copied and I can move a **My_pair** iff **X** and **Y** can be moved. What do I have to ensure that if we don't have implicit generation of copy and move? First try:

```
template<class X, class Y> struct My_pair {
        X first;
        Y second;
        enable_if<has_copy<X>&&has_copy<Y>, Rec(const Rec&) = default>;
        enable_if<has_move<X>&&has_move<Y>, Rec(Rec&&) = default>;
        enable_if<has_copy<X>&&has_copy<Y>, Rec& operator=(const Rec&) = default>;
        enable_if<has_move<X>&&has_move<Y>,Rec& operator=(Rec&&) = default>;
};
```

Thought is "you must be kidding!" My second "but that's not even legal; do I really need to look in the standard t to figure out how to write a simple pair-of-members **struct**?" My third thought was somewhat less polite.

Here is a more correct version:

```
template<class X, class Y> struct My_pair {
```

```
        X first;
        Y second;
        Rec(const (enable_if<has_copy<X>&&has_copy<Y>, Rec>::type&) = default;
        Rec(enable_if<has_move<X>&&has_move<Y>, Rec>::type&&) = default;
        enable_if<has_copy<X>&&has_copy<Y>, Rec>::type&
                operator=(const Rec&) = default>;
        enable_if<has_move<X>&&has_move<Y>,Rec>::type&
                operator=(Rec&&) = default;
};
```

Now, what would happen if a programmer gave up on that mess and simply wrote the first (simple and obvious) version of **My_pair**? A **swap()** of **My_pair** would be (unnecessarily) slow, a **push_back()** of **My_pair** would be (unnecessarily) slow, a **sort()** of **My_pair** (e.g. based on **My_pair<X,Y>::first**) would be (unnecessarily) slow.

A similar performance problem would (by default) occur for every class containing a member of a type that significantly benefits from move, such as

```
struct Indices {
        vector<int> m;
        // operations for conveniently collecting members of m and accessing m
};
```

Returning an **Indices** from a function would be significantly more expensive than returning a plain **vector<int>**. If we teach people to return collections of values from functions by value to avoid messing with free store or out parameters (as I was planning to based on the FCD), use of **Indeces** would become a performance bug.

Unless we want to treat move as an esoteric feature used mostly by expert library implementers and feared and ridiculed as obscure by "ordinary programmers," we need to be able to write something like **My_pair** exactly as the first version or very like it. I would prefer to see move become as common and as well understood as copy. If we cannot achieve that, we should consider eliminating move semantics from the standard (which would imply a most undesirable delay and loss of functionality).

One might argue that the (potential) problems with **My_pair** would be a good reason to use **std::pair** rather than **My_pair**, but programmers do like to write their own types, not every type has an obvious standard library equivalent, and programmers should not have to use specially optimized standard library functions to get good performance.

Copy and move defaults

We have a long-standing ideal of treating copy and move equivalently from a language perspective whenever possible. The reasons are that

- 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

Unless we come up with a new, better, and dramatically different explanation of the relationship between copy and move, this implies that we should not pick fundamentally different defaults for move and copy or invent new syntax for move that doesn't apply to copy.

This implies that allowing copy by default, but not move, and inventing new terse syntax for defaulting move is a dead end.

Treating special operations as invariants

I have called invariants that were unsupported by special operations (copy, move, and destructor) "implicit" and invariants that were indicated by only one of those "partially specified." There may be a possible solution based on that. Consider:

1. Move and copy are generated by default (if an only if their elements move or copy as currently specified in the FCD)
2. If any move, copy, or destructor is explicitly specified (declared, defined, **=default**, or **=delete**) by the user, no copy or move is generated by default.

This (2) is a strengthened version of what is currently in the FCD (e.g., 12.8[8]). Currently, a user-defined copy constructor inhibits generation of a move constructor (and vice verse). Similarly, a user-defined copy assignment inhibits generation of a move constructor (and vice verse). What (2) does is to tie copy constructors, copy assignments, move constructors, move assignments, and destructors together conceptually as part of the notion of the copy operations, move operations, and the destructor defining an invariant.

An aside: I believe that these rule can be specified by a simple local modification of 12.8[8].

These rules would "catch" the classical copy problem (**Vec3**), but not the completely implicit invariant problems (**Vec** and **Vec2**). It would also catch the example that started this round of discussions about implicit move generation:

```
enum PositionState { empty, nought, cross };

class SomeonesClass
{
private:
        std::vector<PositionState>    positions_;

public:
        SomeonesClass(): positions_(9) {}

        SomeonesClass& operator=( SomeonesClass const& other )  {
                for( int i = 0;  i < 9;  ++i )
```

```
                {
                        positions_.at( i ) = other.positions_.at( i );
                }
                return *this;
        }

    // In C++0x implicitly declared: SomeonesClass( SomeonesClass&& other )
};
```

The declaration of the copy assignment would imply that attempts to move assign, move construct, or copy construct a **SomeonesClass** would not compile. To make **SomeonesClass** work exactly as in C++98 would require adding a copy constructor (presumably using **=default**).

So, what do I do if I want to disable moves? We had

```
class Vec {
        vector<int> v;
        int my_int;     // the index of my favorite int stored in v
};
```

which allows copy and move. To disable copy *and* move we can write

```
class Vec1 {
        vector<int> v;
        int my_int;     // the index of my favorite int stored in v
        Vec1(Vec1&&) = delete;          // disable copy and move
};
```

or alternatively

```
class Vec2 {
        vector<int> v;
        int my_int;     // the index of my favorite int stored in v
        ~Vec2() = default;      // disable copy and move
};
```

To get copy but not move is a bit verbose:

```
class Vec3 {
        vector<int> v;
        int my_int;     // the index of my favorite int stored in v
        Vec3(const Vec3&) = default;    // disable copy but not move
        Vec1& operator=(const Vec3&) = default;
};
```

We no longer need the **Vec3(Vec3&&) = delete;** because mentioning a copy operator inhibits implicit generation.

The notation/verbosity problem arise because we have four operations (move and copy constructors and move and copy assignments) that we often want to treat as a whole, occasionally want to treat as two groups (move and copy), and ever-so-rarely want to treat as four individual operations.

Any set of rules that deals with groups of operations will surprise someone who thinks in terms of individual operations (by being more restrictive than expected), whereas any rule that deals only with individual operations will be verbose.

If the use of the signatures is considered too verbose, we could again consider further abbreviations, but I suspect that approach will fail to gain support because of opposition to "late invention."

Other indicators of invariants

It has been pointed out that treating copy, move, and destructors as indicators of the presence of an invariant (in the programmers head) is most appropriate for invariants relating to resource management. That is indeed so and I explored that approach because I consider invariants related to resource management the most interesting and important. What other indicators might we use? The two most obvious are:

- The presence of any constructor
- The absence of public data

The presence of any constructor cannot be the sole indicator of the existence of an invariant (and therefore cannot be used by itself to suppress generation of copy and move). The reason is that many constructors are used simply to provide convenient initialization. For example:

```
struct Point {
        Int x,y;
        Point(int xx, int yy) :x(xx), y(yy) {}
};
```

Here there is no invariant for a **Point**: **x** and **y** can take arbitrary values. The standard library contains examples of that, such as **pair** and **complex**. Obviously, we want to be able to copy **Point**, **pair<int,int>**, and **complex<double>**.

The absence of public data is a good indicator of the presence of an invariant, though not an infallible one. Consider:

```
class Color {
public:
        int get() const;
        void set(int c);
private:
        int col;
};
```

Examples of such simple encapsulated state where copy is necessary and move is either harmless or useful abound.

We might consider the absence of private data a strong indicator of the absence of an invariant: If I can modify all your data arbitrarily, you don't have an invariant that you can trust. The invariant exists only in the programmer's head and is possibly reflected in all code using the class. Unfortunately, even absence of private data is not infallible. I have seen code where data was public (or protected) simply to allow inspection, debug output, etc., yet had an invariant plus a comment "always leave the data in a good state." Some people prefer to rely on trust rather than language-enforced protection.

We might explore using the presence of public data or the absence of private data as an indicator of the absence of an invariant in addition to looking at user-declared move, copy, and destructors. However, I haven't yet had time to explore that alternative fully. For example, a rule base only on a public/private distinction would allow the copy of **My_pair** (good), but not **Color** (bad).

I think that it would be wisest to ignore "invariants" that exist only in the imagination of programmers as non-existent as far as the language design is concerned. The language rules for the use of a class should be based on declarative properties of the class (and not on the comments or the meaning of magic constants in the code).

Special pleading for the standard library

My view is that if it's good enough for the general user, it's good enough for the standard library (but not necessarily vise verse). The ideal is that the standard library is written in C++ and that writers of the standard library should be given no preference over other programmers. I know there are current violations of this ideal, but consider those unfortunate and (in hindsight) avoidable.

There has been claims that **std::pair** is somehow different from an otherwise equivalent classes written by "ordinary users," such as **My_pair** above, by virtue of **std:pair** being in the standard library or (worse) having been in the standard library in 1998, or by virtue of having had copy and move operations explicitly declared. I consider that completely backwards: We can tolerate more verbosity in (standard) library code and possibly even require more stringency of library writers because a library is "written once and used many times" and because library writer must be assumed to be more than average competent. For the end user, we must provide the best possible support for convenience and terseness.

If the standard library facilities, such as **std:pair** "works better" than a programmer's own roughly equivalent types (because they have move operations), programmers will (reluctantly and with little understanding) use the library facilities. This will imply that they by default will get move operations that will break implicit and partial invariants. Thus, we would have imposed an inconvenience (non-generation of move operations) without actually having avoided to problems that caused us to worry about implicit generation of move operations.

What about "old code?"

The suggestion above (suppress implicit generation of copy and move if the users defines a copy, move, or destructor) disallows many questionable C++98 programs, such as **Vec3** (which ought to be broken) and **SomeonesClass** (which you could argue was OK, though I personally consider it rather bad code from a number of perspectives). However, in EWG we decided not to disallow code relying on generated copy operations even if the programmer specified a destructor; there is too much of it "out there" and opinions on what is questionable vary too much. So, I suggest that we only deprecate generation of copy operations where we ban generation of move operations. That way, we can point out that the rules are sane and consistent; we just aren't able to enforce them. I would hope for warnings from compilers.

I think that the most worrying cases are equivalent to **Vec**. For such classes there is an implicit invariant but no "indication" to help the compiler in the form of a user-specified copy constructor. This kind of example occurs (occurred) when a programmer decided that the default copy operations were correct and then (correctly) decided not to mention them because the default copy operations are superior to user-defined ones (e.g. because of ABI issues). In C++0x, a programmer can be explicit by defaulting copy, but that could be considered undesirably verbose. How much code will be broken by the actual use of moved from objects of such types is unknown. I note that even where such types are used, the problem will emerge only in a moved-from object is used.

These compatibility problems must be weighed against the usability problems (e.g. in the "Defaulting notation" section).

I would not oppose the more consistent rules (breaking more code) because I think the immediate effect would be very similar to deprecation: compilers would accept old usage with warnings for a (possibly long) transition period.

I do not consider the current FCD rules "completely broken", but have a strong preference for seeing them improved because they will certainly cause problems that I consider avoidable.

State of moved-from objects

The question of what is guaranteed for a moved-from object is interesting, important, and I think separate from when and how move operations are provided. My position is that I can live with just about any variant of "an unspecified valid state" (as used for the basic guarantee), but that would hate to see a new special ("Zombie") state defined for moved-from objects. For many (most?) classes for which move makes sense there are some obvious "empty" state. Complicating the conceptual framework with some "semi-valid" state would invariably cause confusion and complexity.

Someone (Howard Hinnant, I think) expressed this idea as "'moved-from' has exactly the same meaning as 'after a failed basic guarantee operation' (valid/consistent state, but exact state not specified)."

Conclusions

To have move useful, it must be implicitly generated in many cases. To minimize surprises and bugs, no move operations should be generated for classes with a user-specified copy, move, or destructor. To keep the rules consistent, the generation of copy operations should be deprecated for classes with a user-specified copy, move, or destructor.

Acknowledgements

Thanks to all who made constructive comments in the discussion of move and especially to people who provided difficult code examples.