N2904=09-0094
2009-06-18
Bjarne Stroustrup

Defining default copy and move

Abstract

This note examines the generation of default copy and move operations when elements are combined into an aggregate. It proposes a simple, uniform, safe and efficient scheme. It also examines a couple of problems with the existing rules related to destructors and explicit copy constructors and proposes a remedy that is not 100% compatible.

Introduction

In N2855=09-0045 Doug Gregor and Dave Abrahams identified a serious problem relating to the copy/move for **std::pair** (and by implication for essentially every data structure with two or more elements). N2855 also suggested a solution based on compile-time control of exception throwing. The more I looked at N2855, the more I became convinced that the solution was treating a symptom, rather than a root cause. The problems – but not the solution – was uncannily similar to what we saw when the binding/overloading rules caused us to pick the move for an lvalue in some cases where a class had a move constructor but not a copy constructor [???]. Consequently, this is a proposal based on changing the relationship between move and copy to essentially always consider them together rather than individually when looking to implement a copy construction or copy assignment. The obvious aim is to ensure that

1. *Type safety*: A move constructor is never implicitly applied to an lvalue.
2. *Efficiency*: A copy constructor is never implicitly used for an rvalue if the object is of a type with a move constructor.

This proposal achieves (1) and (2). To simplify library writing, it further provides a slightly stronger guarantee

3. *Simplicity*: A copy constructor is never implicitly used to implement a move constructor

I do not propose any changes to exception handling rules.

The rest of this note is organized like this

- Explain the basic idea using a terse notation
- Give examples of implications
- Discuss key design issues
- Summarize the proposed rules
- List the proposed WP changes

Basic idea

First consider every built-in type to have both a copy and a move constructor. Doing so allows us to treat all types in the same way and to generalize rules from individual types to aggregates (I used the word "aggregate" in its general meaning of a collection of elements, rather than its C/C++ restricted meaning). The obvious implementation of a move of an object of a built-in type is a copy.

For exposition, let **C** mean "has a copy constructor", let **M** mean "has a move constructor", and let **CM** mean "has a move and a copy constructor". We can then express a set of rules for composing two types into an aggregate (e.g. using **std::pair**, array, **struct**, **class**, etc.). The idea is that depending on what moves and copies are supplied by the member types we can (or cannot) generate appropriate move and copy constructors for the aggregate:

1. **CM**+**CM** => **CM**; that is, if both components have both **C** and **M**, the aggregate also has **C** and **M**
2. **CM**+**C** => **C**
3. **CM**+**M** => **M**
4. **C**+**C** => **C**
5. **M**+**M** => **M**
6. **M**+**C** => nothing

Rules 2-6 can be summarized as "we can generate a move or copy constructor if and only if both elements has that constructor"

Exactly what "noting" means in rule 6 will be explained below, but it must be a variant of "you can't (by default) copy or move an aggregate of those two elements."

Rule 2 and 6 are discussed below because it would seem that if you want to move, but only have a copy constructor, you could use the copy constructor with only loss of efficiency.

Examples

Let's apply the rules to a set of example to get an idea of their implications.

```
pair<complex<double>,int>      //  has M and C
```

std::complex gets both **M** and **C** (move and copy constructors) because both of its elements (of type **double** and **int**) have both **M** and **C**. Consequently, **std::pair** gets both **M** and **C** because both of its elements have them. Incidentally, this example shows how we generalize the rules for to element to aggregates of N elements.

```
pair<unique_ptr<double>,int>  // has M but not C
```

We can't copy a **unique_ptr**, so obviously we can't (by default) copy something that contains a **unique_ptr**, but we can move both a **unique_ptr** and an **int**.

The rule for **std::pair** is simply a special case of the general rule for classes:

```
struct S1 { complex<double> m1; int m2; };      // S1 has M and C
```

```
struct S2 { unique_ptr<double> m1; int m2; };   // S2 has M but not C
```

Consider an example that we will consider in detail below:

```
struct NM {      // C but not M
        NM(NM&&) = delete;          // no move
        NM(const NM&) = default;    // default copy
};
pair<unique_ptr<double>,NM>         // has neither M nor C
```

We can copy one element, but not the other, and move one element but not the other, so we can neither copy nor move the aggregate. Here are some examples demonstrating the "we can generate a move or copy constructor if and only both elements has that constructor" rule of thumb:

```
pair<complex<double>,NM>    // has C but not M
tuple<complex<double>,int,unique_ptr<double>>       // has M but not C
tuple<pair<complex<double>,int>,pair<unique_ptr<double>,int>>       // has M but not C
struct S3 {
        complex<double> m1;         // MC
        int m2;                     // MC
        unique_ptr<double> m3;      // M
        int m4;                     // MC
};      // has M but not C
```

This shows that the rules are easily applied to **std::tuple** and to class members in general; you just apply them recursively.

Using a copy to move

Why should we prohibit using a copy to implement a move? In other words, why don't we have a rule

2. **CM**+**C** => **CM** // not proposed

That is, taking the view that a copy is simply a move where we don't modify the source and synthesize the move from a copy of the first element and a move of the second?

I suggest three concerns:

1. *Efficiency*: For many types, people expect a move to be efficient compared to a copy, but if no provisions have been made to move an element, you get copy costs when it is part of an aggregate.
2. *Type*: a copy and a move have radically different argument types; much confusion has been cause by this and I would fear surprises. In terms of type-safety as such, I don't see a problem, but I fear the complexity and confusion.
3. *Exceptions*: moves tend not to throw, but can't provide the strong guarantee (since they may – and often do – modify their source), whereas copies often throw (they acquire resources), but can typically provide the strong guarantee.

That latter point was what prompted N2855. It is really hard (arguably impossible) to implement important operations, such as **push_back()**, simply, efficiently, and providing the strong (exception) guarantee if a move might be a copy in disguise. Never implicitly using a copy to implement a move avoids these problems.

But what if I (explicitly) implemented a move constructor using a copy of an element? I would have to do so explicitly because the rule for implicit generation is the strict **CM**+**C** => **C**. Unless I caught every exception thrown by the copy, I would have written bad code that violates a (so far unsystematic) rule that move constructors should not throw. In particular, a throwing move would violate the requirements for many standard library components. I think that N2855 demonstrates that that rule is needed and I suggest that it should be an exact parallel to the rule that destructors should not throw. I suspect that statically enforcing the "don't throw" rule for move constructors and destructors would do more harm than good. Either way, that question is orthogonal to the issue of how we specify move operations.

What to do when we can't copy or move

Consider

6. **M**+**C** => nothing

What should "nothing" mean? I see two plausible alternatives

- A compile-time error when you compose the aggregate
- A compile-time error if you try to copy or move the aggregate

I suggest the latter because some aggregates are never copied or moved. That is, the aggregate should have its copy and move constructors deleted (i.e., as if **=delete** had been explicitly used).

When can we generate copies and moves?

For historical reasons going back to 1979 changes from K&R C to Classic C, a copy is by default generated. This is convenient in many simple cases, but also causes problems, such as slicing and bad interactions with destructors. Taking a safer and more conservative approach, we could decide not to generate a move unless a programmer explicitly requests so. That radical difference in the rules for copy

and move would be illogical, but it is not obvious whether we can do anything about it: There is a lot of code "out there."

Before trying to craft rules, let's consider what would be ideal:

1. Have uniform rules for copy and move
2. Don't generate a copy or move for an aggregate containing pointers (they may represent ownership)
3. Don't generate copy or move for an aggregate with a destructor (if it has a destructor, something has a meaning that requires cleanup – and that typically means that copying is hazardous)
4. Don't generate a copy or move if a class is a base class of another class (because then the derived class could be sliced).
5. Allow explicit **=delete** and **=default** for copy and move

Consider (3). Could we do better by defining move as default copy followed by destruction of the source? Not really: A copy could impose unexpected overheads and might even throw (as pointed out in N2855).

Consider (4): We can do something about that only be observing a slicing operation. For example:

```
Struct B { int b; };
Struct D : B { int d; };
Void f(B);
D d;
f(d);     // oops slicing
```

I'm not proposing to do anything about that just now. The slicing problem seems a bit separate from the rest and I mention is only for completeness.

We have three basic choices for approximating these ideals:

1. Define the rules to be uniform and not to generate problematic moves or copies (not compatible even with C)
2. Define the rules to be uniform and generate moves that give us the same problems as we now have for copies (compatible and replicates problems).
3. Define non-uniform rules that eliminate problematic moves (only; compatible but doesn't replicate problems)

For reasons of compatibility and convenience in simple cases, I will not consider the radical simplifications of just not implicitly providing default copies or moves.

Consider some examples:

// Rule 1 (uniform and incompatible):

```
struct S0 { int x; }          // copy and move

struct S1 { int* p; };        // no copy or move (C++98 and C incompatibility)

struct S2 { int x; ~S2(); };  // no copy or move (C++98 incompatibility)

struct S3 { int* p; ~S3 (); }; // no copy or move (C++98 incompatibility)
```

// Rule 2 (uniform, compatible, and error-prone):

```
struct S0 { int x; }          // copy and move

struct S1 { int* p; };        // copy and move (error prone)

struct S2 { int x; ~S2(); };  // copy and move (error prone)

struct S3 { int* p; ~S3(); }; // copy and move (most likely an error)
```

// Rule 3 (non-uniform, compatible, and error-prone):

```
struct S0 { int x; }          // copy and move

struct S1 { int* p; };        // no move; copy ok (error prone)

struct S2 { int x; ~S2(); };  // no move; copy ok (error prone)

struct S3 { int* p; ~S3(); }; // no move; copy ok (most likely an error)
```

Having been bitten by bad implicitly defined copies, I find Rule 1 *very* tempting. However, I don't think it is feasible – it would break a lot of code. Having non-uniform rules for move and copy would be confusing would lead to unnecessary errors (for copy or for both copy and move), so consider what we might feasibly define "problematic moves and copies." Note that

1. *any* restriction on generated copies will break some code somewhere.
2. such broken code will be detected and flagged by the compiler – there will be no silent breakage.
3. the (compile-time) error will not occur just for a declaration but only for attempts to copy.
4. the code can be easily repaired by a **=default**.
5. in some cases the code deserves to be broken either because it hides actual errors or because it is a maintenance hazard (not that we should expect every user to appreciate the advantages of breaking their code, however horrid)

So the question is how can we define "problematic moves and copies" to catch the most errors with the lowest number of false positives (errors for correct C++98 code)? I propose this:

// Rule 4:

```
struct S0 { int x; }          // copy and move
```

```
struct S1 { int* p; };          // copy and move

struct S2 { int x; ~S2(); };    // no copy or move

struct S3 { int* p; ~S3(); };   // no copy or move
```

That is, we don't implicitly copy or move classes with destructors:

4. Define the rules to be uniform and not to generate moves or copies for classes with a destructor.

This is a very simple rule which is compatible with C and with much C++98 practice. It would catch a class of common errors. A variant of Rule 4 would delete copy only if a class had both a destructor and a pointer member, but I'm not sure if that extra complexity is worth the extra compatibility that it would buy us. So I propose Rule 4, but would accept the variant if someone could demonstrate that the variant broke significantly less old code or left reasonable code unbroken where Rule 4 did not.

In the unlikely case that someone needs a reminder, here is the canonical example of what I want to protect against:

```
class V {
        double* elem;
        int sz;
public:
        V(int s) : sz(s), elem(new double[s]) { }
        ~V() { delete[] elem; }
        // default copy
        // …
};

V  f()  // innocent (looking) code
{
        V v1(100);
        V v2 = v1;
        return v1; // crash here (if we are lucky, otherwise memory corruption)
}
```

Multiple deletions of a free store allocation can be really nasty. Despite almost 30 years of warnings, many people still make that mistake. We have to weigh the pain and outrage caused by such examples against the outcry against C++0x breaking code.

Copy and move assignments

Copy and move assignments should be handled exactly equivalently to copy and move constructors.

Defaults and deletes

For a move, **=default** means member-wise move of each element, if possible, otherwise a compile-time error is immediately given. This differs from implicitly using the default in that we get no error from the implicit default where we get an error only if used. That's the same rules as for a default copy.

If an implicit move cannot be generated, e.g., because a member has a **=deleted** move, a user can explicitly define one. Whether a user can write a good move for such a type is less certain.

Questions:

- Should we generate a move for a class for which a programmer has explicitly defined a copy (i.e., not **=default**)? In that case, the programmer might have defined a non-standard semantics that might affect move. Suggested answer: yes.
- Should we generate a move for a class with a deleted copy? Suggested answer: yes. After all, the copy might be deleted exactly because moving was intended.

My reason for not distinguishing between user-defined copy operators and default ones when considering whether to generate a move is that I suspect most such copy operations are redundant (should have been **=default**) or simply do "house keeping" (e.g. updates a count). A user that does not want the default move can **=delete** it or write another.

What should be the semantics of a move of an object of a built-in type? It should be a copy. One might consider zeroing out a moved object, but that would imply an overhead that is inconsistent with a basic idea of moves: to be simpler and more efficient than copies. Obviously, a debug option might be for a move to zero out pointers.

Explicit

A copy constructor can be **explicit** or not. However, a default (generated) constructor is always implicit: 12.3.1[3]: "A non-explicit copy-constructor (12.8) is a converting constructor. An implicitly-declared copy constructor is not an explicit constructor; it may be called for implicit type conversions." Unless, I'm mistaken, this could be seen as giving rise to an oddity:

```
struct X {
        X() {}
        explicit X(const X&) { }
};

struct Y {
        Y() {}
        Y(const Y&) { }
};

struct Z {
        X x;
```

```
        Y y;
        // default copy
};

int main()
{
        X a;
        X b = a;        // error: explicit copy constructor (just right)
        Y aa;
        Y bb = aa;
        Z aaa;
        Z bbb = bbb;    // is this right?
}
```

It appears that **explicit** disappears when we combine copy operations. One could argue that we explicitly copied the X in Z's generated copy constructor, effectively writing

```
Z::Z(const Z& a) :x(a.x), y(a.y) { }
```

Thus implicitly using **X::X()** explicitly. Is that what we want?

I think we have three alternatives:

1. Generate a copy of an aggregate as non-explicit independently of whether an element has explicit copy or not (status quo).
2. Generate a copy of an aggregate as an **explicit** copy if an element has an explicit copy and non-explicit otherwise.
3. Don't generate a copy for an aggregate if an element has an explicit copy.

I propose (2), but whichever choice is made, the same choice should be made for move. My reason for preferring (2) is that I think that having **explicit** quietly disappear when we (implicitly) combine objects violate reasonable assumptions (there is nothing explicit about an implicitly generated constructors) and could be argued to be a type violation (in the example above, I explicitly said that an B may not be implicitly copied).

Naturally, this will break some code, but not quietly, and I guess that explicit copy operations are rare and their use in classes with default copy operations is even rarer.

Proposal summary

The proposal has the following parts

1. All built-in types are assumed to have move constructors and their semantics are that of copy constructors
2. By default, an aggregate of elements has an implicitly defined move constructor if every element has a move constructor, a copy constructor if every element has a copy constructor

3. A move constructor or an copy constructor are not generated for a class with a destructor
4. If a move constructor or a copy constructor is not explicitly defined and cannot be (implicitly) generated, it is considered **=deleted**.
5. Move constructors (like copy constructors) can be explicitly declared or explicitly defined **=deleted** or **=default**.
6. Rules for move assignments and copy assignments are analogous to those of move constructors and copy constructors.
7. A move constructor or a move assignment may not exit through an exception being thrown (as for a destructor).
8. An implicitly defined move constructor for an aggregate of elements is **explicit** if any of its elements have their move constructor **explicit**. Similarly for a copy constructor.

The proposal

12.8 Copying class objects [class.copy]: Add a case to the list of cases where an implicitly declared copy constructor is deleted (at the end of [5]):

- X has a destructor

Also add a new paragraph:

An implicitly-declared copy constructor is explicit if the copy constructor of any of its members or bases is explicit.

Add a case to the list of cases where an implicitly declared copy assignment is deleted (at the end of [10]):

- X has a destructor

Add a new section **12.10 Moving objects [class.move]** (this section is intended to be analogous to 12.8):

An object of a built-in type is assumed to have a move constructor.

A class object can be moved in two ways, by initialization or by assignment. Conceptually, these operations are done by a move constructor and a move assignment operator.

A non-template constructor for a class X is a move constructor if its sole parameter is of type X&&.

If the class definition does not explicitly declare a move constructor, one is declared implicitly. The implicitly-declared move constructor for class X will have the form

X::X(X&&)

If

- each direct or virtual base class B of X has a move constructor
- each non-static member of X has a move constructor

An implicitly-declared move constructor is an inline public member of its class. An implicitly-declared move constructor for a class X is defined as deleted if X has:

- a variant member with a non-trivial move constructor and X is a union-like class
- a non-static data member or base has a deleted move constructor
- X has a destructor

A move constructor for class X is trivial if it is not user-provided and if

- class X has no virtual functions or virtual classes
- the move constructor of each member and class is trivial

Otherwise the move constructor is non-trivial.

An implicitly-declared move constructor is explicit if the move constructor of any of its members and bases is explicit.

A move constructor may not return by throwing an exception.

A non-template assignment operator for a class X is a move assignment operator if its sole parameter is of type X&&.

If the class definition does not explicitly declare a move assignment operator, one is declared implicitly. The implicitly-declared move assignment operator for class X will have the form

X& X::operator=(X&&)

If

- each direct or virtual base class B of X has a move assignment operator
- each non-static member of X has a move assignment operator

An implicitly-declared move assignment operator is an inline public member of its class. An implicitly-declared move assignment operator for a class X is defined as deleted if X has:

- a variant member with a non-trivial move assignment operator and X is a union-like class
- a non-static data member or base has a deleted move assignment operator
- X has a destructor

A move assignment operator for class X is trivial if it is not user-provided and if

- class X has no virtual functions or virtual classes
- the move assignment operator of each member and class is trivial

Otherwise the move assignment operator is non-trivial.

An implicitly-declared move assignment operator is explicit if the move assignment operator of any of its members and bases is explicit.

A move assignment operator may not return by throwing an exception.