Implicit Move Must Go

Document Number: N3153=10-0143
Document Date: 2010-10-17
Author: Dave Abrahams <dave@boostpro.com>

What is Implicit Move?

In N2855, the authors described a surprising effect that occurs when "legacy" C++03 types are combined in the same object with move-enabled C++0x types: the combined type could acquire a throwing move constructor. At that time, we didn't have a way to implement `vector::push_back` and a few other important strong-guarantee operations in the presence of a throwing move.

Several independent efforts were made to deal with this problem. One approach, while not a complete solution, shrank the problem space considerably and had many other benefits: **implicitly generate move constructors and move assignment operators** when not supplied by the user, in much the same way that we do for copy constructors and copy assignment operators. That is the implicit move feature, and it was voted into the working paper this past spring.

Further Background

Another effort, spurred on by Rani Sharoni's comments at C++Next finally yielded a complete, standalone solution to the problem, a form of which was eventually also adopted by the committee. Nobody attempted to "repeal" the implicit generation of move operations from the standard document, not least because of those "many other benefits" I alluded to earlier. But now we have a problem that we hadn't anticipated.

The Problem

Back in August, Scott Meyers posted to comp.lang.c++ about a problem where implicit generation of move constructors could break C++03 class invariants. For example, the following valid C++03 program would be broken under the current C++0x rules (and indeed, is broken in g++4.6 with `--std=c++0x`) by the implicit generation of a move constructor:

```
#define _GLIBCXX_DEBUG
#include <iostream>
#include <vector>
```

```
struct X
{
    // invariant: v.size() == 5
    X() : v(5) {}

    ~X()
    {
        std::cout << v[0] << std::endl;
    }

 private:
    std::vector<int> v;
};

int main()
{
    std::vector<X> y;
    y.push_back(X()); // X() rvalue: copied in C++03, moved in C++0x
}
```

The key problem here is that in C++03, `X` had an invariant that its `v` member always had 5 elements. `X::~X()` counted on that invariant, but the newly-introduced move constructor moved from `v`, thereby setting its length to zero.

Tweak #1: Destructors Suppress Implicit Move

Because rvalues are generally "about to be destroyed," and the broken invariant was only detected in `X`'s destructor, it's tempting to think that we can "tweak" the current rules by preventing the generation of implicit move constructors when a user-defined destructor is present. However, the following example would still break (and breaks under g++4.6 with `--std=c++0x`):

```
#define _GLIBCXX_DEBUG
#include <algorithm>
#include <iostream>
#include <vector>

struct Y
{
    // invariant: values.size() > 0
    Y() : values(1, i++) {}
    Y(int n) : values(1,n) {}

    bool operator==(Y const& rhs) const
    {
        return this->values == rhs.values;
    }

    int operator[](unsigned i) const
    { return values[i]; }

 private:
    static int i;
    std::vector<int> values;
};

int Y::i = 0;
```

```
int main()
{
    Y ys[10];
    std::remove(&ys[0], &ys[0]+10, Y(5));
    std::cout << ys[9][0];
};
```

In C++03, there's no way to create a `Y` with a zero-length `values` member, but because `std::remove` is allowed to use move operations in C++0x, it can leave a moved-from `Y` at the end of the array, and that could be empty, causing undefined behavior in the last line.

About the use of `std::remove` in these examples

In C++03, `std::remove` eliminates values from a range by *assigning over them* (technically it can also use swap, but let's assume assignment for now). Since it can't actually change sequence *structure*, it assigns over the unwanted elements with values from later in the sequence, pushing everything toward the front until there's a subrange containing only what's desired, and returns the new end iterator of that subrange . For example, after `remove`ing `0` from the sequence `0 1 2 0 5`, we'd end up with `1 2 5`, and then `0 5`—the last two elements of the sequence would be unchanged.

In C++0x, we have move semantics, and `std::remove` has permission to use *move assignment*. So in C++0x, we'd end up with `1 2 5 0 x` at the end of the sequence, where `x` is the value left over after moving from the last element—if the elements are `int`s, that would be `5`, but if they are `BigNum`s, it could be anything. Similar properties are shared by `std::remove_if` and `std::unique`.

There's another way moved-from values can be exposed to C++03 code running under C++0x: an algorithm such as `sort` can throw an exception while shuffling elements, and you can then observe a state where not everything has been moved back into place. Showing that just makes for more complicated examples, however.

Tweak #2: Constructors Suppress Implicit Move

It's also tempting to think that at least in classes without a user-defined constructor, we could safely conclude that there's no intention to maintain an invariant, but that reasoning, too, is flawed:

```
#define _GLIBCXX_DEBUG
#include <iostream>
#include <vector>

// An always-initialized wrapper for unsigned int
struct Number
{
    Number(unsigned x = 0) : value(x) {}
    operator unsigned() const { return value; }
 private:
    unsigned value;
};

struct Y
{
    // Invariant: length == values.size().  Default ctor is fine.

    // Maintains the invariant
    void resize(unsigned n)
    {
        std::vector<int> s(n);
        swap(s,values);
        length = Number(n);
    }

    bool operator==(Y const& rhs) const
    {
        return this->values == rhs.values;
    }

    friend std::ostream& operator<<(std::ostream& s, Y const& a)
    {
        for (unsigned i = 0; i < a.length; ++i)
            std::cout << a.values[i] << " ";
        return s;
    };

 private:
    std::vector<int> values;
    Number length;
};

int main()
{
    std::vector<Y> z(1, Y());

    Y a;
    a.resize(2);
    z.push_back(a);

    std::remove(z.begin(), z.end(), Y());
    std::cout << z[1] << std::endl;
};
```

In this case, the invariant that `length == values.size()` was established by the well-understood default-construction behavior of subobjects, but implicit move generation has violated it.

Tweak #3: Private Members Suppress Implicit Move

It's also tempting to think that we can use private members to indicate that an invariant needs to be preserved; that a "C-style `struct`" is not encapsulated and has no need of protection. But the members of a privately-inherited struct are effectively encapsulated and private with respect to the derived class. It is not uncommon to see members moved into an implementation detail struct, which is then used as a base class:

```
// Modified fragment of previous example.  Replace the definition
// of Y with this code.

namespace detail
{
  struct Y_impl
  {
      std::vector<int> values;
      Number length;
  }
}

struct Y : private Y_impl
{
    void resize(unsigned n)
    {
        std::vector<int> s(n);
        swap(s,values);
        length = Number(n);
    }

    bool operator==(Y const& rhs) const
    {
        return this->values == rhs.values;
    }

    friend std::ostream& operator<<(std::ostream& s, Y const& a)
    {
        for (unsigned i = 0; i < a.length; ++i)
            std::cout << a.values[i] << " ";
        return s;
    };
};
```

Real-World Examples

Of course these examples are all somewhat contrived, but they are not unrealistic. We've already found classes in the (new) standard library—`std::piecewise_linear_distribution::param_type` and `std::piecewise_linear_distribution`—that have been implemented in exactly such a way as to expose the same problems. In particular, they were shipped with g++4.5, which

had no explicit move, and not updated for g++4.6, which did. Thus they were broken by the introduction of implicitly-generated move constructors.

Summary

I actually like implicit move. It would be a very good idea in a new language, where we didn't have legacy code to consider. Unfortunately, it breaks fairly pedestrian-looking C++03 examples. We could continue to explore tweaks to the rules for implicit move generation, but each tweak we need to make eliminates implicit move for another category of types where it could have been useful, and weakens confidence that we have analyzed the situation correctly. And it's very late in the standardization process to tolerate such uncertainty.

Conclusions

It is time to remove implicitly-generated move operations from the draft. That suggestion may seem radical, but implicit move was proposed very late in the process on the premise that it "treated…the root cause" of the exception-safety issues revealed in N2855. However, it did not treat those causes: we still needed `noexcept`. Therefore, implicitly-generated move operations can be removed without fundamentally undermining the usefulness or safety of rvalue references.

The default semantics of the proposed implicit move operations are still quite useful and commonly-needed. Therefore, while removing implicit generation, we should retain the ability to produce those semantics with "`= default`." It would also be nice if the rules allowed a more concise way to say "give me all the defaults for move and copy assignment," but this paper offers no such proposal.