J16/00-0009
WG21/N1232
March 6, 2000
J. Stephen Adamczyk
jsa@edg.com

The core language `auto_ptr` problem

In core issue 84, I described a remaining core language problem with `auto_ptr`. I raised that issue near the end of 1998, and I was hoping that newer formulations of `auto_ptr` might have managed to avoid the trouble spot. Unfortunately, the version I got from Greg Colvin recently (code at the end of this paper) still has the same problem.

I will try to explain the problem, and then recommend changes to deal with it. My recommendation, however, is effectively to do nothing, i.e., to leave `auto_ptr` partially broken.

The Problem

In Greg's example, the line that doesn't work with EDG's compiler is the one marked "ERROR here":

```
Base::sink(source());    // ERROR here
```

This calls a static member function, passing an rvalue of type `auto_ptr<Derived>` to a parameter of type `auto_ptr<Base>`.

Argument passing is done by copy-initialization, so this example is effectively the same as the declaration

```
auto_ptr<Base> temp = (an rvalue of type auto_ptr<Derived>);
```

This is handled by 8.5p14, in the bullet beginning "Otherwise (i.e., for the remaining copy-initialization cases)," because the source and destination types are not the same type or base/derived types (note that `auto_ptr<Derived>` is not a derived class of `auto_ptr<Base>`). That text sends us to 13.3.1.4 to enumerate the constructors and conversion functions that are candidate functions:

- The converting constructors of `auto_ptr<Base>`. There are two of these: a non-template with parameter list `(auto_ptr<Base> &)`, and a template with parameter list `(auto_ptr<Y> &)`. Neither of these is viable. The parameter in each case is a reference to non-const, so it must be bound to an lvalue. The source expression, however, is an rvalue, and there is no conversion function that could convert it to an lvalue.
- The conversion functions of `auto_ptr<Derived>` that can convert to `auto_ptr<Base>` or a derived class thereof. There is one of these: a conversion function template that converts to `auto_ptr<Y>`. This turns out to be viable, and is selected.

So far, so good. Now, however, back in 8.5p14, we get to the words “The result of the call (which is the temporary for the constructor case) is then used to direct-initialize, according to the rules above, the object that is the destination of the copy-initialization.”

This is the “copy” part of the copy-initialization, the part that is skipped when copy constructor elision is done. Generally, a programmer would expect that the copy would be elided, and would therefore expect to be able to ignore it. A programmer might expect that the result of the conversion function would be passed directly as the argument to the `sink` function.

That turns out to be overly simplistic. In the normal case, the copy is done (or would be done) by calling a copy constructor. In this case, however, the result of calling the conversion function is an rvalue, and therefore it cannot be copied by the available constructors, whose parameters have reference-to-nonconst type. So we end up using another conversion function. Here’s the analysis:

We want to direct-initialize an entity of type `auto_ptr<Base>` from an rvalue of type `auto_ptr<Base>`. In 8.5p14, this falls under the bullet beginning “If the initialization is direct-initialization”, because the source and destination types are the same. We are sent to 13.3.1.3 to enumerate the candidate functions, which are the constructors of `auto_ptr<Base>`[1] :

- The constructor with parameter list `(auto_ptr<Base> &)` and the template constructor with parameter list `(auto_ptr<Y> &)` cannot bind to an rvalue and therefore are not viable.
- The constructor with parameter list `(Base * = 0)` is not viable because there is no way to convert to that parameter type.
- The non-template constructor with parameter list `(auto_ptr_ref<Base>)` is viable (maybe; more on this below): we can use a conversion function to convert the `auto_ptr<Base>` rvalue to an `auto_ptr_ref<Base>` rvalue, then use the implicitly-declared copy constructor of `auto_ptr_ref<Base>` to pass the by-value parameter. This is, once again, copy-initialization.

That means that to pass the argument for the original example, we call

- a conversion function from `auto_ptr<Derived>` to `auto_ptr<Base>`,
- a conversion function from `auto_ptr<Base>` to `auto_ptr_ref<Base>`,
- a copy constructor to copy an `auto_ptr_ref<Base>`, and
- a constructor to make an `auto_ptr<Base>` from an `auto_ptr_ref<Base>`.

This bothers me, because I have always believed that it is a rule of C++ that no more than one user-defined conversion will be called to do an implicit conversion, and this case calls either two or three, depending on how you count.

By the way, because 12.8p15 says copy constructor elision is allowed when a “copy constructor” is called, and not on an arbitrary copy, a standard-conforming compiler is not allowed to elide the copy in the above and must call at least the two conversion functions and the final constructor.

---

1. The suggested change for core issue 152 would make this just the *converting* constructors. That has no effect on the analysis here.

The Recommended Solution

I recommend that we make changes to reaffirm the common-law rule that at most one user-defined conversion is allowed in an initialization.

If you squint the right way, it might be possible to read the standard that way already. Derek Inglis has pointed out that 13.3.1p6 says "because only one user-defined conversion is allowed in an implicit conversion sequence, …", and there's similar wording in 13.3.3.1. His position is that it's already clear that at most one user-defined conversion is allowed in cases like the above. I think it's clear that only one user-defined conversion is allowed in an implicit conversion sequence, but the rules for copy-initialization lead to two passes through overload resolution, and each of those involves a separate implicit conversion sequence, thus (unfortunately) allowing two user-defined conversions. I agree with Derek's desired outcome; I just don't agree that the standard already says that.

The wording that needs to be changed to address this is 13.3.3.1p4:

> In the context of an initialization by user-defined conversion (i.e., when considering the argument of a user-defined conversion function; see 13.3.1.4, 13.3.1.5), only standard conversion sequences and ellipsis conversion sequences are allowed.

What I'd really like to see this changed to is

> When considering the argument of a user-defined conversion function that is a candidate by 13.3.1.3 when invoked for copy-initialization, or by 13.3.1.4, 13.3.1.5, or 13.3.1.6 in all cases, only standard conversion sequences and ellipsis conversion sequences are allowed.

(Paragraph 3 of the same section would have to be reworded to make it apply to all cases not covered by paragraph 4, because the simple "Except in the context of an initialization by user-defined conversion" would no longer be specific enough.)

This is what, I believe, we should have done when we were working on the standard. It differs from the current wording in that it does not allow conversion functions to be called on the arguments of a constructor used to do a copy-initialization from the same type or a derived type, e.g.,

```
Base b = Base();
Base bb = Derived();
```

Declarations like the above would be accepted if and only if there is a constructor (which might be a template) that would allow the copying. No conversion functions would be tried. Note that, until the `auto_ptr` trick, it was not thought that it would ever be necessary or desirable to call a conversion function if you already had the right type. `auto_ptr`, however, deliberately has no copy constructor that can copy and rvalue, and has a conversion function to the helper class `auto_ptr_ref` so that an `auto_ptr` rvalue can be "copied" by converting it to `auto_ptr_ref` and back to `auto_ptr`.

This, as I say, it what I'd like to see done. However, that change would close the loophole exploited by auto_ptr and make auto_ptr not work at all. I believe it would break nothing at all except auto_ptr and any other classes developed in the last couple of years by programmers who saw the auto_ptr trick and thought it is really "cool". I believe that's not very much code, given that I implemented the loophole in EDG's compiler at the end of 1998, and before that I never got any complaints about this aspect.

But I suspect that the committee will prefer a more modest change, which simply closes the secondary use of the loophole:

> When considering the argument of a user-defined conversion function that is a candidate by 13.3.1.3 when invoked for the copying of the temporary in the second step of a class copy-initialization, or by 13.3.1.4, 13.3.1.5, or 13.3.1.6 in all cases, only standard conversion sequences and ellipsis conversion sequences are allowed.

This would make the example quoted above invalid, and enforce the general principle that at most one user-defined conversion is invoked implicitly, while still preserving most of the auto_ptr functionality.

Code for auto_ptr (from Greg Colvin via e-mail, Dec. 11, 1999)

```
namespace std {

   class auto_ptr_base {
      template<typename Y> friend struct auto_ptr;
      template<typename Y> friend struct auto_ptr_ref;
      void* p;
   };

   template <typename X> class auto_ptr_ref {
      template<typename Y> friend struct auto_ptr;
      auto_ptr_ref(X* p, auto_ptr_base& r) : r(r), p(p) {}
      X* release() const { r.p = 0; return p; }

      auto_ptr_base& r;
      X* const p;
   };

   template<typename X> struct auto_ptr : auto_ptr_base  {
      typedef X element_type;

      explicit auto_ptr(X* px =0) throw() { p = px; }
      auto_ptr(auto_ptr& r) throw() { p = (void*)r.release(); }
      template<typename Y> auto_ptr(auto_ptr<Y>& r) throw() {
         X* px = r.release();
         p = (void*)px;
```

```
      }
      auto_ptr& operator=(auto_ptr& r) throw() {
         reset(r.release());
         return *this;
      }
      template<typename Y> auto_ptr& operator=(auto_ptr<Y>& r)
                                                      throw() {
         reset(r.release());
         return *this;
      }
      ~auto_ptr() { delete get(); }

      X& operator*() const throw() { return *get(); }
      X* operator->() const throw() { return get(); }

      X* get() const throw() { return static_cast<X*>(p); }
      X* release() throw() { X* px = get(); p = 0; return px; }
      void reset(X* px=0) throw()
                 {if (px != get()) delete get(), p = (void*)px; }

      auto_ptr(auto_ptr_ref<X> r) throw() {
         p = (void*)r.release();
      }
      auto_ptr& operator=(auto_ptr_ref<X> r) throw() {
         reset(r.release());
         return *this;
      }
      template<typename Y> operator auto_ptr_ref<Y>() throw() {
         return auto_ptr_ref<Y>(get(),*this);
      }
      template<typename Y> operator auto_ptr<Y>() throw() {
         return auto_ptr<Y>(release());
      }
   };
}



////////////////////////////////////////////////////////////////
#include <stdio.h>
#include <memory>
using namespace std;

struct MemoryTracker
{
    MemoryTracker(void* newMem)
```

```
        : memory(newMem), next(list) { list = this; }

    static bool StopTracking(void* oldMem) {
        for ( MemoryTracker* p = list; p != 0; p = p->next ) {
            if ( p->memory == oldMem ) {
                p->memory = 0;
                return true;
            }
        }
        ++nFailure;
        puts( “Bad delete” );
        return false;
    }

    static int CheckAllDeleted() {
        for ( MemoryTracker* p = list; p != 0; p = p->next ) {
            if ( p->memory != 0 )
                ++nFailure, puts(“not deleted”);
        }
        return nFailure;
    }

private:
   void* memory;
   MemoryTracker* next;
   static int nFailure;
   static MemoryTracker* list; // linked list of all memory
                               // trackers
};

int MemoryTracker::nFailure = 0;
MemoryTracker* MemoryTracker::list = 0;

struct Base
{
   virtual ~Base() {}    // So we can delete a Derived through a
                         // Base*

   // To test passing auto_ptr<Derived> as auto_ptr<Base>
   static void sink(auto_ptr<Base>) {}

   void* operator new(unsigned n) {
      void* p = ::operator new(n);
      new MemoryTracker(p);
      return p;
   }
```

```
   void operator delete(void* p) {
      if (MemoryTracker::StopTracking( p ));
         ::operator delete(p);
   }

   char dummy; // force this class to occupy space
};

// A dummy base class
struct ForceOffset {
   char dummy; // force this class to occupy space
};

// Trying to force Derived and Base to have different addresses
struct Derived : ForceOffset, Base {
   // To test passing auto_ptr<Derived> as auto_ptr<Derived>
   static void sink(auto_ptr<Derived>) {}

   char dummy; // force this class to occupy space
};

auto_ptr<Derived> source() {
   return auto_ptr<Derived>(new Derived);
}

void test() {
   // Test functionality with no conversions
   auto_ptr<const Derived> p(source());
   auto_ptr<const Derived> pp(p);
   Derived::sink(source());
   p = pp;
   p = source();

   // Test functionality with Derived->base conversions
   auto_ptr<const Base> q(source());
   auto_ptr<const Base> qp(p);
   Base::sink(source());    // ERROR here
   q = pp;
   q = source();
}

int main() {
   test();
   return MemoryTracker::CheckAllDeleted();
}
```