Doc. No.: J16/97-0090R1= WG21/N1128R1
Date: November 14, 1997
Project: Programming Language C++
Reply to: Bill Gibbons <bill@gibbons.org>
Greg Colvin <greg@imrgold.com>

Fixing `auto_ptr`.

The *auto_ptr* specified in CD-2 has proved unpopular and dangerous, primarily because the *const* arguments to its copy operations make it easy to inadvertently damage an *auto_ptr* via a *const* reference, and because the non-owning pointer left behind by a copy is an open invitation to dangling references. The *auto_ptr&* arguments to the copy constructor and assignment operator were not *const* in the CD-1 *auto_ptr*, but were made *const* to allow *auto_ptr* values to be passed to and returned from functions. The C++ language now allows a more effective solution.

We propose to restore the CD-1 *auto_ptr* semantics by:

- removing *const* from the arguments to all copy operations and from the release() function;
- restoring the pointer-zeroing effect of release();
- restoring the reset() function; and
- adding conversion functions and a private auxiliary class to allow *auto_ptr* rvalues to convert to lvalues.

Draft text to replace 20.4.5 follows.

20.4.5 Template class `auto_ptr`

1 Template `auto_ptr` holds a pointer to an object obtained via new and deletes that object when it itself is destroyed (such as when leaving block scope 6.7).

2 Template `auto_ptr_ref` holds a reference to an `auto_ptr`. It is used by the `auto_ptr` conversions to allow `auto_ptr` objects to be passed to and returned from functions.

```
namespace std {
  template<class X> class auto_ptr {
     template<class Y> struct auto_ptr_ref {};
  public:
    typedef X element_type;

  // 20.4.5.1 construct/copy/destroy:
    explicit auto_ptr(X* p=0) throw();
    auto_ptr(auto_ptr&) throw();
    template<class Y> auto_ptr(auto_ptr<Y>&) throw();
    auto_ptr& operator=(auto_ptr&) throw()
    template<class Y> auto_ptr& operator=(auto_ptr<Y>&) throw();
    ~auto_ptr() throw();

  // 20.4.5.2 members:
    X& operator*() const throw();
    X* operator->() const throw();
    X* get() const throw();
    X* release() throw();
    void reset(X* p=0) throw();

  // 20.4.5.3 conversions:
    auto_ptr(auto_ptr_ref<X>) throw();
    template<class Y> operator auto_ptr_ref <Y>() throw();
    template<class Y> operator auto_ptr<Y>() throw();
  };
}
```

3 The `auto_ptr` provides a semantics of strict ownership. An `auto_ptr` owns the object it holds a pointer to. Copying an `auto_ptr` copies the pointer and transfers ownership to the destination. If more than one `auto_ptr` owns the same object at the same time the behavior of the program is undefined.

20.4.5.1 `auto_ptr` constructors

```
explicit auto_ptr(X* p =0) throw();
```

1 **Postconditions:** `*this` holds the pointer `p`.

```
auto_ptr(auto_ptr& a) throw();
```

2 **Effects:** Calls `a.release()`.

3 **Postconditions:** `*this` holds the pointer returned from `a.release()`.

```
template<class Y> auto_ptr(auto_ptr<Y>& a) throw();
```

4 **Requires:** `Y*` can be implicitly converted to `X*`.

5 **Effects:** Calls `a.release()`.

6 **Postconditions:** `*this` holds the pointer returned from `a.release()`.

```
auto_ptr& operator=(auto_ptr& a) throw();
```

7 **Requires:** The expression `delete get()` is well formed.

8 **Effects:** `reset(a.release())`.

9 **Returns:** `*this`.

```
template<class Y> auto_ptr& operator=(auto_ptr<Y>& a) throw();
```

10 **Requires:** `Y*` can be implicitly converted to `X*`. The expression `delete get()` is well formed.

11 **Effects:** `reset(a.release())`.

12 **Returns:** `*this`.

```
~auto_ptr() throw();
```

13 **Requires:** The expression `delete get()` is well formed.

14 **Effects:** `delete get()`.

20.4.5.2 `auto_ptr` members

```
X& operator*() const throw();
```

1 **Requires:** `get() != 0`

2 **Returns:** `*get()`

```
X* operator->() const throw();
```

3 **Requires:** `get() != 0`

4 **Returns:** `get()`

```
X* get() const throw();
```

5 **Returns:** The pointer `*this` holds.

```
X* release() throw();
```

6 **Returns:** `get()`

7 **Postconditions:** `*this` holds the null pointer.

```
void reset(X* p=0) throw();
```

8 **Effects:** If `get() != p` then `delete get()`.

9 **Postconditions:** `*this` holds the pointer p.

20.4.5.3 `auto_ptr` conversions

```
auto_ptr(auto_ptr_ref<X> r) throw();
```

1 **Effects:** Calls `p->release()` for the `auto_ptr p` that `r` holds.

2 **Postconditions:** `*this` holds the pointer returned from `release()`.

```
template<class Y> operator auto_ptr_ref <Y>() throw();
```

3 **Returns:** An `auto_ptr_ref<Y>` that holds `*this`.

```
template<class Y> operator auto_ptr<Y>() throw();
```

4 **Effects:** Calls `release()`.

5 **Returns:** An `auto_ptr<Y>` that holds the pointer returned from `release()`.

Analysis of Conversion operations

There are four cases to consider: direct-initialization and copy-initialization (8.5/14) for both same-type initialization and base-from-derived initialization.

(1) Direct-initialization, same type, e.g.

```
auto_ptr<int> source();

auto_ptr<int> p( source() );
```

This is considered a direct call to a constructor of *auto_ptr<int>*,using overload resolution. There is only one viable constructor:

```
auto_ptr<int>::auto_ptr(auto_ptr_ref<int>)
```

which is callable using the conversion

```
auto_ptr<int>::operator auto_ptr_ref<int>()
```

which should be selected when operator overloading tries to convert type *auto_ptr<int>* to *auto_ptr_ref<int>*.

Overload resolution succeeds. No additional copying is allowed, so the copy constructor need not be callable.

(2) Copy-initialization, same type, e.g.

```
auto_ptr<int> source();
void sink( auto_ptr<int> );

main() {
    sink( source() );
}
```

According to 8.5/14:

> If the initialization is direct-initialization, or if it is copy-initialization where the cv-unqualified version of the source type is the same class as, or a derived class of, the class of the destination, constructors are considered...

So this case is handled the same as the direct-initialization case.

(3) Direct-initialization, base-from-derived, e.g.

```
struct Base {};
struct Derived : Base {};
auto_ptr<Derived> source();

auto_ptr<Base> p( source() );
```

This is similar to (1); in this case, the viable constructor is:

```
auto_ptr<Base>::auto_ptr(auto_ptr_ref<Base>)
```

which is callable using the conversion

```
auto_ptr<Derived>::operator auto_ptr_ref<Base>()
```

which should be selected when operator overloading tries to convert type *auto_ptr<Derived>* to *auto_ptr_ref<Base>*.

Overload resolution succeeds. No additional copying is allowed, so the copy constructor need not be callable.

(4) Copy-initialization, base-from-derived, e.g.

```
struct Base {};
struct Derived : Base {};
auto_ptr<Derived> source();
void sink( auto_ptr<Base> );

main() {
    sink( source() );
}
```

This case is not similar to (2), because the sentence quoted above from 8.5/14 does not apply. So there must be a conversion function (operator or constructor) from the argument type to the parameter type, and it will be used to initialize a temporary variable. Note that this initialization process does not involve use of a copy constructor:

The user-defined conversion so selected is called to convert the initializer expression into a temporary, whose type is the type returned by the call of the user-defined conversion function, with the cv-qualifiers of the destination type.

The parameter type is *auto_ptr<Base>*, so there must be a conversion from *auto_ptr<Derived>* to *auto_ptr<Base>.* The constructor

```
auto_ptr<Base>::auto_ptr<Derived>(auto_ptr<Derived> &)
```

does not work because the argument is an rvalue. But the conversion function

```
auto_ptr<Derived>::operator auto_ptr<Base>()
```

does work. The result of calling this conversion function is a temporary - no copy constructor is needed.

Once the temporary has been created, the draft says:

> The object being initialized is then direct-initialized from the temporary according to the rules above.

This direct-initialization is case (1) which works.

At no time in any of these four cases is the implementation allowed to make an unnecessary copy of an *auto_ptr* object. Therefore it does not matter that the copy constructor does not work on rvalues.