X3J16/97-0012
WG21/N1050
January 27, 1997
J. Stephen Adamczyk
jsa@edg.com

Binding a reference to the result of a conversion function

In 8.5.3 [dcl.init.ref] the WP says

A reference to type “cv1 T1” is initialized by an expression of type “cv2 T2” as follows:

- If the initializer expression is an lvalue (but not an lvalue for a bit-field), and
  - “cv1 T1” is reference-compatible with “cv2 T2,” or
  - T2 is a class type, and the initializer expression can be implicitly converted to an lvalue of type “cv3 T3,” where “cv1 T1” is reference-compatible with “cv3 T3” (this conversion is selected by enumerating the applicable conversion functions (_over.match.ref_) and choosing the best one through overload resolution (_over.match_)), then

  the reference is bound directly to the initializer expression lvalue in the first case, and the reference is bound to the lvalue result of the conversion in the second case. In this case the reference is said to *bind directly* to the initializer expression.

Note that the requirement that the initializer expression be an lvalue applies also to the second sub-bullet case. That means the following is an error:

```
struct A {};
struct B {
  operator A&();
};
B f();
A &r = f(); // Error, original expression is an rvalue
```

The logic of that lvalue restriction, as I remember it, is that it’s dangerous to allow an rvalue (which by definition has a limited lifetime) to implicitly turn into an lvalue (which has, potentially, a longer lifetime). It makes it too easy to produce dangling references, e.g., if the conversion function just returns a reference to some subobject in the object passed to it. Of course, this may just be a style issue — programmers should perhaps be told that if they return a reference from a conversion function, it should be to something allocated and not to a subobject of what’s passed in.

Existing compilers give no error on the above, for what that’s worth. (EDG, g++ 2.6.3, Sun 3.0.1, MSVC++ 4.2, and cfront 3.0.2 all accept it.)

Of greater concern, perhaps, is the case where the reference is to const:

```
struct A {};
struct B {
  operator A&();
};
B f();
const A &r = f(); // Valid, but copies
```

This case is valid according to the WP, but the implementation is *required* to copy the result of the conversion function to a temporary, and bind the reference to that. There is no option, as there is in some similar cases (see 8.5.3 paragraph 5), of an implementation choice to bind directly to the result or copy and bind to a temporary.

This extra copy seems not to be existing practice, and therefore may be surprising and/or undesirable. EDG just implemented the lvalue restriction, and ran into problems with one valarray implementation because the extra copy sliced a polymorphic object and as a consequence a pure virtual function was called. And the conversion function in that case did do an allocation, so there was no dangling-reference problem.

I raised this issue in a core-reflector message, and all the responses expressed the opinion that the lvalue requirement was inappropriate.

So I propose to eliminate it by changing the above-quoted text (from dcl.init.ref) to:

A reference to type “cv1 T1” is initialized by an expression of type “cv2 T2” as follows:

- If the initializer expression
  - is an lvalue (but not an lvalue for a bit-field) and “cv1 T1” is reference-compatible with “cv2 T2,” or
  - has a class type (i.e., T2 is class type) and can be implicitly converted to an lvalue of type “cv3 T3,” where “cv1 T1” is reference-compatible with “cv3 T3” (this conversion is selected by enumerating the applicable conversion functions (_over.match.ref_) and choosing the best one through overload resolution (_over.match_)), then

  the reference is bound directly to the initializer expression lvalue in the first case, and the reference is bound to the lvalue result of the conversion in the second case. In these cases the reference is said to *bind directly* to the initializer expression.