X3J16/96-0021
WG21/N0839

# Eliminating the Class Rvalue Standard Conversion

J. Stephen Adamczyk (jsa@edg.com)
Edison Design Group, Inc.

January 28, 1996

## Introduction

When I rewrote the standard conversions chapter, I added a standard conversion from an rvalue of a derived class to an rvalue of a base class (4.12). I did so because one of my reviewers felt the conversion was needed for casts. Later, casts were defined in terms of initialization, which did away with the erstwhile reason for the standard conversion. At the Austin meeting, I explained the history and asked for a vote on removing the standard conversion. The straw vote was inconclusive (18 for, 18 against, and 13 abstaining), so I proposed no formal motion and the standard conversion stayed in the WP.

I'm more than ever convinced that the standard conversion is a bad idea and that it should be removed. I believe the support for retaining the standard conversion was based on a desire to see a certain class of examples "work," so I'm proposing a different change to make those acceptable, and once again proposing removal of the standard conversion.

## Why the standard conversion is bad

The class rvalue standard conversion doesn't look like the other standard conversions:

- All of the other standard conversions work on built-in types. This one works on user-defined types only.[1]
- No other standard conversion involves calling a function. Worse yet, in this case it may actually take overload resolution to select the function to be called (though the WP is silent on that count).

The fact that a user-defined copy constructor gets called also brings up the possibility that this standard conversion opens a loophole in the rule that at most one user-defined conversion is used for any implicit conversion. Or rather, it seems to open such a loophole. I think we're actually saved from that[2] because the standard conversion doesn't actually get used in the cases where you would think it might:

[1]enumeration types can be converted to integral types, but that's because we sometimes want them to be user-defined types, and sometimes just fancy integral types.

[2]Though it's hard to be positive; yet another reason to get rid of this standard conversion.

```
struct B {};
struct D : public B {};
D d;
B b = d;  // Nope, not here
```

It only comes up in really obscure cases (see below). And when it does come up in those cases, since it calls the copy constructor, a copy is made. But the copy ends up seeming like a superfluous step.

In summary: When it seems that it ought to apply, it doesn't. When it does apply, it does the wrong thing. And always, it doesn't look like the rest of the standard conversions.

## The problem case

The standard conversion survived because of an obscure example:

```
struct X {};
struct Y : public X {};
int operator==(X, X);
struct Z {
  operator Y();
};
Z z1, z2;
int i = (z1 == z2);
```

When considering the `operator==` function with respect to `z1 == z2`, the initialization of an argument of the function reduces to the initialization

```
X param = z1;
```

which, because `X` is not the same class as `Z` or a base class thereof, becomes equivalent to

```
X temp(z1);
```

followed by a copy constructor call to copy `temp` to `param`.

This final initialization can be made to work by converting `z1` to an rvalue of type `Y` by use of the conversion function, followed by a derived-to-base standard conversion to convert that rvalue to type `X`. Without the standard conversion, it is ill-formed.

*Should* that example be valid? The strongest argument on the positive side is that the example is well-formed if the operator function is changed to take its parameters by reference:

```
struct X {};
struct Y : public X {};
int operator==(const X&, const X&);
struct Z {
  operator Y();
};
Z z1, z2;
int i = (z1 == z2);
```

This seems so similar to the original formulation that one is inclined to say that if one works, the other ought to as well. But are they really the same? In the pass-by-value case, the argument is copied, and therefore sliced, and its polymorphic behavior changes. In the reference case, whether or not a copy is done is implementation-defined, but either way the polymorphic behavior is preserved.

(A reality check: what do existing compilers do? EDG's front end, g++ 2.6.3, Sun 3.0.1, Borland 4.5, and Microsoft Visual C++ 4.0 give an error. cfront 3.0.2 and Watcom 10.0 accept the example.)

At the discussion in Austin, some said that if getting rid of the standard conversion gets rid of some sneaky cases of implicit slicing, by disallowing examples like this, then let's get rid of the standard conversion. Others felt that it was more important that the pass-by-value and pass-by-reference cases be consistent.

At the time, I didn't have an opinion on that issue, but I do now. Under the principle that C++ should provide mechanism and not legislate style, I believe the example as originally written, with pass-by-value parameters, should be allowed. The programmer should be free to write the operator function in the most appropriate way. If he or she chooses the pass-by-value approach, we should assume that he or she does so with the understanding that slicing can occur. Perhaps the programmer knows that slicing will not be a problem. Whatever. Slicing of the result of a conversion function is only one of the possible kinds of slicing that an operator function written with pass-by-value parameters could encounter, and by no means the simplest:

```
struct X {};
struct Y : public X {};
int operator==(X, X);
Y y1, y2;
int i = (y1 == y2);  // Slicing on the argument passing
```

So we should leave it to the programmer to make the right decision.

## Further analysis

But does that mean that the standard conversion is necessary? Let's look more closely at the example. Here's what happens:

1. The conversion function `Z::operator Y` is called with the argument `z1`. It returns a temporary of type `Y`.
2. To do the standard conversion, the copy constructor `X::X(const X&)` is called to copy the temporary to another temporary of type `X`, thus doing the slicing.
3. The copy constructor `X::X(const X&)` is called again to copy that second temporary into the parameter.

The first call of the copy constructor seems pretty silly. One would expect that an implementation would use the copy constructor elision rule to eliminate it.

To produce the list of steps above, I used the principle that an initialization like[3]

```
X param = z1;
```

is rewritten as

```
X param(X(z1));
```

This is embodied in 8.5 [dcl.init] paragraph 12, fourth bullet, third sub-bullet:

[3]Recall that "="-style initialization is used for argument passing.

> Otherwise (i.e., for the remaining copy-initialization cases), a temporary of the destination type is created. User-defined conversion sequences that can convert from the source type to the destination type are enumerated (_over.match.user_), and the best one is chosen through overload resolution (_over.match_). The user-defined conversion so selected is called to convert the initializer expression into the temporary. If the conversion cannot be done or is ambiguous, the initialization is ill-formed. The object being initialized is then direct-initialized from the temporary according to the rules above. [Footnote: Because the type of the temporary is the same as the type of the object being initialized, this direct-initialization, if well-formed, will use a copy constructor (_class.copy_) to copy the temporary.] In certain cases, an implementation is permitted to eliminate the temporary by initializing the object directly; see _class.temporary_.

Now, when a copy constructor is used like this, its parameter is a reference to the class being copied, and it is therefore capable of binding to an object of a derived class. So another rewrite of the "="-form initialization above that could be considered is

```
X param(Y(z1));
```

That is, you convert to a temporary of a derived class, and then copy (slice) that temporary to do the "="-form initialization. This gets the desired result, in the optimized form, without recourse to a "standard conversion." The derived-to-base shift, and the associated slicing, is done as a normal consequence of the way copy constructors and reference parameters work. This makes a lot more sense to me than a derived-to-base standard conversion.

And that's the proposal: Currently, in the definition of copy-initialization, the temporary created always has the same type as the class object being initialized, and only conversion functions that convert to that class type are considered. With the proposed change, conversion functions that convert to derived classes of the type of the object being initialized would also be considered. If one of those is selected by overload resolution, the temporary would have the derived class type. That temporary is then copied to the object being initialized by calling the copy constructor.

## A related case

Let's also look at a related case that is covered by a different part of the WP:

```
struct X {};
struct Y : public X {};
int operator==(const X&, const X&);
struct Z {
  operator Y&();
};
Z z1, z2;
int i = (z1 == z2);
```

Here, the conversion function is capable of creating an lvalue of type `Y`. Since the parameters of the `operator==` function could be made to bind to such an lvalue, one might expect that this example is well-formed. But the current WP disallows it. Why? 8.5.3 [dcl.init.ref] 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
  - the initializer expression can be implicitly converted (_conv_) to an lvalue of type "cv3 T1," where cv3 is the same cv-qualification as, or lesser cv-qualification than, cv1, then

etc.

In this example, this considers only conversions to `X`, not conversions to the derived class `Y`, so the conversion function is not found.

(Another reality check: g++ 2.6.3, Sun 3.0.1, and Microsoft Visual C++ 4.0 give an error. EDG's front end, cfront 3.0.2, Borland 4.5, and Watcom 10.0 accept the example.)

Again, I believe this case ought to be well-formed. The suggested change to the WP is to also consider conversions to derived class lvalues for this reference binding case.

## WP changes

Summary: Delete the standard conversion (4.12, [conv.class]) and some references to it; split 13.3.1.3 [over.match.user] into three paragraphs, one dealing with user-defined conversions for copy-initialization of a class, one dealing with initializations of nonclasses from classes by use of a conversion function, and one dealing with initialization by conversion function for direct reference binding; change 8.5 [dcl.init] and 8.5.3 [dcl.init.ref] to refer to these new paragraphs.

(These changes are relative to 95-0185/N0785 except where indicated.)

In 3.8 [basic.life], around paragraph 6, delete the words indicated here in italics (they were added for the Santa Cruz WP):

> If an lvalue-to-rvalue conversion (_conv.lval_) is applied to such a reference, the program has undefined behavior; if the original object will be or was of a non-POD class type, the program has undefined behavior if:
>
> - the reference is used to access a non-static data member or call a non-static member function of the object, or
> - *the reference is implicitly converted (_conv.class_) to a base class type, or*
> - the reference is used as the operand of a `static_cast` (_expr.static.cast_) (except when the conversion is to `char&`), or
> - the reference is used as the operand of a `dynamic_cast` (_expr.dynamic.cast_) or as the operand of `typeid`.

In 4 [conv], paragraph 1, second bullet, delete the words indicated here in italics:

> Zero or one conversion from the following set: integral promotions, floating point promotion, integral conversions, floating point conversions, floating-integral conversions, pointer conversions, pointer to member conversions, *base class conversions,* and boolean conversions.

Delete 4.12 [conv.class] in its entirety (it defines the standard conversion).

Change 8.5 [dcl.init], paragraph 12, fourth bullet, third sub-bullet, to the following:

> Otherwise (i.e., for the remaining copy-initialization cases), a temporary is created. User-defined conversion sequences that can convert from the source type to the destination type or a derived type thereof are enumerated (_over.match.copy_), and the best one is chosen through overload resolution (_over.match_). 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. If the conversion cannot be done or is ambiguous, the initialization is ill-formed. The object being initialized is then direct-initialized from the temporary according to the rules above. [Footnote: Because the type of the temporary is the same as the type of the object being initialized, or a derived class thereof, this direct-initialization, if well-formed, will use a copy constructor (_class.copy_) to copy the temporary.] In certain cases, an implementation is permitted to eliminate the temporary by initializing the object directly; see _class.temporary_.

In 8.5 [dcl.init], paragraph 12, fifth bullet, change the second sentence as indicated here in boldface:

> Otherwise, if the source type is a (possibly cv-qualified) class type, conversion functions are considered. The applicable conversion functions are enumerated **(_over.match.conv_)**, and the best one is chosen through overload resolution (_over.match_). The user-defined conversion so selected is called to convert the initializer expression into the object being initialized. If the conversion cannot be done or is ambiguous, the initialization is ill-formed.

In 8.5.1 [dcl.init.aggr], paragraph 11, change the first sentence as indicated here in boldface:

> All **implicit** type conversions **(_conv_)** are considered when initializing the aggregate member with an initializer from an initializer-list.

Change 8.5.3 [dcl.init.ref], paragraph 6, second sub-bullet, to the following:

> T2 is a class type, and the initializer expression can be implicitly converted to an lvalue of type "cv3 T3," where T3 is the same type as T2, or is a derived type thereof, and cv3 is the same cv-qualification as, or lesser cv-qualification than, cv1 [Footnote: This requires a conversion function (_class.conv.fct_) returning a reference type.] (this conversion is selected by enumerating the applicable conversion functions (_over.match.ref_) and choosing the best one through overload resolution (_over.match_)), then

In 13.3 [over.match], paragraph 3, replace the last two bullets with the following and change "five" to "seven" in the first line:

Overload resolution selects the function to call in seven contexts within the language:

- ...
- invocation of a constructor for direct-initialization (_dcl.init_) of a class object (_over.match.ctor_);
- invocation of a user-defined conversion for copy-initialization (_dcl.init_) of a class object (_over.match.copy_);
- invocation of a conversion function for initialization of an object of a nonclass type from an expression of class type (_over.match.conv_); and
- invocation of a conversion function for initialization of a temporary to which a reference (_dcl.init.ref_) will be directly bound (_over.match.ref_).

In Table 9, in 13.3.3.1.1 [over.ics.scs], delete the line for base class conversion.

Delete 13.3.1.3 [over.match.user] and replace it with the following (inserted after 13.3.1.4 [over.match.ctor] because the order is better that way):

13.3.1.4 Copy-initialization of class by user-defined conversion [over.match.copy]

Under the conditions specified in _dcl.init_, as part of a copy-initialization of an object of class type, a user-defined conversion can be invoked to convert an initializer expression to the type of the object being initialized. Overload resolution is used to select the user-defined conversion to be invoked. Assuming that "cv1 T" is the type of the object being initialized, with T a class type, the candidate functions are selected as follows:

- The converting constructors (_class.conv.ctor_) of T are candidate functions.
- When the type of the initializer expression is a class type "cv S", the conversion functions of S and its base classes are considered. Those that are not hidden within S and yield type "cv2 T2", where T2 is the same type as T or is a derived class thereof, and where cv2 is the same cv-qualification as, or lesser cv-qualification than, cv1, are candidate functions. Conversions functions that return "reference to T" return lvalues of type T and are therefore considered to yield T for this process of selecting candidate functions.

In both cases, the argument list has one argument, which is the initializer expression. [Note: this argument will be compared against the first parameter of the constructors and against the implicit object parameter of the conversion functions.]

13.3.1.5 Initialization by conversion function [over.match.conv]

Under the conditions specified in _dcl.init_, as part of an initialization of an object of nonclass type, a conversion function can be invoked to convert an initializer expression of class type to the type of the object being initialized. Overload resolution is used to select the conversion function to be invoked. Assuming that "cv1 T" is the type of the object being initialized, and "cv S" is the type of the initializer expression, with S a class type, the candidate functions are selected as follows:

- The conversion functions of S and its base classes are considered. Those that are not hidden within S and yield type "cv2 T" or a type that can be converted to type "cv2 T" via a standard conversion sequence (_over.ics.scs_), for any cv2 that is the same cv-qualification as, or lesser cv-qualification than, cv1, are candidate functions. Conversion functions that return a nonclass type "cv2 T" are considered to yield cv-unqualified T for this process of selecting candidate functions. Conversions functions that return "reference to T" return lvalues of type T and are therefore considered to yield T for this process of selecting candidate functions.

The argument list has one argument, which is the initializer expression. [Note: this argument will be compared against the implicit object parameter of the conversion functions.]

13.3.1.6 Initialization by conversion function for direct reference binding [over.match.ref]

Under the conditions specified in _dcl.init.ref_, a reference can be bound directly to an lvalue that is the result of applying a conversion function to an initializer expression. Overload resolution is used to select the conversion function to be invoked. Assuming that "cv1 T" is the underlying type of the reference being initialized, and "cv S" is the type of the initializer expression, with S a class type, the candidate functions are selected as follows:

- The conversion functions of S and its base classes are considered. Those that are not hidden within S and yield type "reference to cv2 T2", where T2 is the same type as T or is a derived class thereof, and where cv2 is the same cv-qualification as, or lesser cv-qualification than, cv1, are candidate functions.

The argument list has one argument, which is the initializer expression. [Note: this argument will be compared against the implicit object parameter of the conversion functions.]

The following text from the end of the old 13.3.1.3 [over.match.user] is moved to after paragraph 5 of 13.3.1 [over.match.funcs]:

Because only one user-defined conversion is allowed in an implicit conversion sequence, special rules apply when selecting the best user-defined conversion (_over.match.best_, _over.best.ics_). [Example:

```
class T {
public:
        T();
        // ...
};
class C : T {
public:
        C(int);
        // ...
};
T a = 1;                 // ill-formed: T(C(1)) not tried
```

–end example]

In 13.3.3.2, paragraph 4, remove the following bullets (they describe cases of the derived-to-base standard conversion and are therefore no longer needed):

conversion of C to B is better than conversion of C to A,

conversion of B to A is better than conversion of C to A.