X3J16/96-0017
WG21/N0835

# Conversion Functions in Overload Resolution

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

January 26, 1996

## Introduction

This document proposes a small change regarding the treatment of conversion functions in overload resolution. The change makes some cases work the way programmers expect them to, and brings the Working Paper closer to existing practice.

## The status quo

Consider the following example:

```
struct B {
  operator int();
};
struct D : public B {
  operator long();
};
main () {
  D d;
  int i = d;  // ??
}
```

In the line marked with question marks, `A::operator int` and `D::operator long` compete in overload resolution as ways of converting `d` to type `int`.

In the current working paper description, the `this` parameter of `A::operator int` requires a derived-to-base standard conversion, whereas the `this` parameter of `D::operator long` has an exact match with the argument/object `d`. Therefore, `D::operator long` is chosen. The comparison of the result types of the conversion functions with the `int` type being initialized would happen later in the overload resolution process, but we never get that far. The better match on the argument conversions ends the process right there.

## Why a change is needed

This selection of the `operator long` function is surprising to programmers. Examples like this have come up on the core reflector several times, with general cries of "this can't be right!" Furthermore, it's not at all the existing practice. I tried the above test case on the C++ compilers we have at EDG, and found that EDG's front end, cfront 3.0.2, g++ 2.6.3, Sun 3.0.1, Borland

4.5, Watcom 10.0, and Microsoft Visual C++ 1.51 all select the `operator int`. (Microsoft Visual C++ 4.0 selects the `operator long`, but perhaps it was changed to match the Working Paper. Sorry, guys.)

I don't believe the behavior mandated by the WP was chosen deliberately. In fact, I remember noting that this particular behavior differed from what EDG had implemented and from what cfront did, but I guessed there might be other compilers that did it the other way. And, as Jerry Schwarz has pointed out many times, we didn't care what the precise behavior is in boundary cases, so long as it is specified. It turns out, however, that there is consensus both on how this should work and on how existing compilers implement it. And that consensus doesn't match the WP.

## The suggested change

As it happens, in the years since that decision was made, a similar issue came up as part of namespaces, and the discussion of that problem led to a resolution that, in my opinion, should be adopted in this case as well. Specifically, 7.3.3 [namespace.udecl] contains the following paragraph:

> For the purpose of overload resolution, the functions which are introduced by a using-declaration into a derived class will be treated as though they were members of the derived class. In particular, the implicit this parameter shall be treated as if it were a pointer to the derived class rather than to the base class. This has no effect on the type of the function, and in all other respects the function remains a member of the base class.

The proposal is that the same thing be made true of conversion functions, i.e., from the point of view of the `this` parameter in overload resolution, they would be considered to be members of the class of the object being converted. Therefore there would be no derived-to-base standard conversion in the above example, and the functions would both proceed to the next step, where `A::operator int` would be selected because the conversion from its return type to `int` is better than the conversion from the return type of `D::operator long`.

Note that this change would *not* mean that the `this` parameter would always be an exact match in the general case: cv-qualification differences would still be significant in choosing one conversion function over another.

## Working Paper changes

In 13.3.1 [over.match.funcs], paragraph 4, add after the "example" sentence:

> For conversion functions, the function is considered to be a member of the class of the implicit object argument for the purpose of defining the type of the implicit object parameter. For non-conversion functions introduced by a using-declaration into a derived class, the function is considered to be a member of the derived class for the purpose of defining the type of the implicit object parameter.