Doc No: X3J16/92-0110
WG21/N0187
Date: October 29, 1992
Project: Programming Language C++
Reply To: Kim Coleman
coleman@apple.com

Ambiguities in Overload Resolution Rules

Kim Coleman
coleman@apple.com
(408) 862-7319

Introduction

This paper attempts to summarize ambiguities and open questions about the current overloaded function resolution rules for the Core WG. Some of these issues are drawn from the x3j16-core reflector, others from my own experience or from discussions with others. This paper does not necessarily attempt to solve these problems. This document is intended to evolve as more problems and solutions are identified.

The following notation is used for references to the Working Paper: C.S:P where C is a chapter number, S is a section or subsection number, and P is a paragraph number.

Conversion Sequence Shortening

This discussion is a summary of x3j16-core-1084-6,1088,-90, and 1092. According to 13.2:6,

> For a given actual argument, no sequence of conversions will be considered that contains more than one user-defined conversion or that can be shortened by deleting one or more conversions into another sequence that leads to the type of the corresponding formal argument of any function in consideration. Such a sequence is called a *best-matching* sequence.

Rather than simply stating that a sequence which can be shortened is a worse match than the shorter sequence, this paragraph indicates that it is not even considered. The wording gave rise to some confusion and should probably be changed. Gavin Koch started the discussion with the following example:

```
struct Z {
  operator int();
};
struct A { };
struct B { };

void g(int, A);
void g(double,B);

void f()
{
  Z z;
```

```
  B b;
  g( z, b );
}
```

Gavin's reading of 13.2:6 led him to believe that there was no match for the call of g because g(double,B) is not even considered due to the shorter conversion sequence for g(int, A). It was rightly pointed out by Andy that g(int,A) is thrown out before we start to consider best-matching sequences because there is no possible conversion from Z to A (§ 13.2:1). Thus, the only function available for consideration is g(double,B) and there is no shorter conversion sequence to consider. However, there is a difference between not considering a sequence at all and treating it as a "worse" match. Consider the other type of conversion sequence rejected by 13.2:6, one "that contains more than one user-defined conversion".

```
struct A {  };

struct Z {
  Z(A);
  operator int();
};

void g(int);

void f()
{
  A a;
  g(a);                 // no match
}
```

The call to g is illegal *even though there is only one function under consideration* because it requires two user defined conversions. If a conversion sequence requiring more than user-defined conversion were simply considered a worse match than a sequence requiring only once user-defined conversion, the call would be legal.

For the case of shortened conversion sequence, it may be argued that this differentiation is just semantic hair splitting since there will always be at least two sequences under consideration and it is not possible to end up with an empty set for a given argument due just to this rule. Nevertheless, it is somewhat confusing and not clear that the draft needs to be quite so extreme in it's treatment of conversion sequences that can be shortened. Scott Turner suggested changing the wording of 13.2:6 so that such sequence were simply not a better match than shorter sequences. Possible new wording might be:

> For a given actual argument, no sequence of conversions will be considered that contains more than one user-defined conversion. If a sequence can be shortened by deleting one or more conversions into another sequence that leads to the type of the corresponding formal argument of any function in consideration, the shortened sequence is considered the *best-matching* sequence.

Using Const-ness as a Tie-breaker

In x3j16-core-1093, Jerry Schwarz presents the following example:

```
class A { };
void f(A&);              // variant 1
void f(const A&);        // variant 2

class B : public A  { };

... B b;  f(b); ...
```

The possible conversion sequences are:

(1) B& --[std conv]--> A&
(2) B& --[std conv]--> A& --[triv conv]-->const A&
(3) B& --[triv conv]--> const B& --[std conv]--> const A&

Sequence 2 is discarded in favor of sequence (1), as per 13.2:6. However, this still leaves an apparent tie between sequences (1) and (3). The question is whether or not the fact that sequence (3) involves a conversion to const can be used a tie-breaker. The relevant selections from the Working Paper are at the end of 13.2:8 and matching rule [1] in 13.2:11.

> Sequences of trivial conversions that differ only in order are indistinguishable. Note that functions with arguments of type T, const T, volatile T, and const volatile T& accept exactly the same set of values. Where necessary, const and volatile are used as tie-breakers as described in rule [1] below.

> [1] Exact match: Sequences of zero or more trivial conversions are better than all other sequences. Of these, those that do not convert T* to const T*, T* to volatile T*, T& to const T&, or T& to volatile T& are better than those that do.

Due primarily to the sentence in 13.2:8, it is not clear whether const and volatile may be used as a tie-breaker in sequences of non-trivial conversions as well. Jerry suggested either modifying 13.2:11, rule [3] to handle this special case or adopting something along the lines of the following general rule:

> "Apply a general rule that says 'If one sequence is obtained from another by dropping consts' then the sequence without const is better."

In x3j16-core-1095, Scott Turner supports the adoption of such a rule and further suggests removing the offending sentence from 13.2:8. He presented the following example which can also benefit from Jerry's new rule:

```
class A { };
class C { };
void f(C&);
void f(const A&);

class B : public A, public C { };

... B b;  f(b); ...
```

Leading to following possible sequences of conversions:

(1) B& --[std conv]--> C&
(2) B& --[std conv]--> A& --[triv conv]--> const A&
(3) B& --[triv conv]--> const B& --[std conv]--> const A&

13.2:8 as it currently stands may be taken to say sequence (1) is preferred over (2) and (3) because it does not involve a conversion to const, but Scott rightly points out that this seems a very weak reason for preferring (1). Jerry's proposed rule does not allow any of the above conversions to be preferred over the others and hence Scott's example becomes ambiguous.

Jerry's rule needs more anaylsys and rewording. Do you drop all consts? What if the target type is const, does it make any sense to drop that last const? What examples might be broken by this?

Ordering of Conversion Sequences

Also in x3j16-core-1095, Scott Turner suggests extending what 13.2:8 says about sequences of trivial conversions differing only in order:

> Sequences of trivial conversions that differ only in order are indistinguishable.

Scott used the following example to illustrate:

```
class A { };
void f(const A&);

class B : public A { };

... B b;  f(b); ...
```

Leading to the following possible conversion sequences:

(1) B& --[std conv]--> A& --[triv conv]--> const A&
(2) B& --[triv conv]--> const B& --[std conv]--> const A&

Intuitively, this example appears ambiguous, but there isn't anything in the Working Paper currently that clearly leads to this conclusion. Scott suggests extending 13.2:8 to cover this example. While I agree that there is nothing that clearly makes this example ambiguous, there also is nothing in the Working Paper that allows one of these conversion sequences to be preferred over the other. Since neither conversion sequence can be considered a best-matching sequence, the call is ambiguous. I don't think there is a need to change the Working Paper for such cases.

Promotion Rules Incomplete?

13.2:8, rule [2] defines a match with promotions as:

> Of sequences not mentioned in [1], those that contain only integral promotions, conversions from float to double, and trivial conversions are better than all others.

The commentary in ARM states that this rule "exists to allow promotions to be preferred over standard conversions." I may be missing something obvious, but I have never understood why

this rule includes the conversion from float to double, but says nothing about conversions from float or double to long double. Consider the following example:

```
void f(double);
void f(long double);

float val;
f(val);
```

Under the current rules, f(double) is unambiguously called because it is a match with promotions while f(long double) is only a match with a standard conversions. This seems very arbitrary. I hope this is just an oversight.

The ANSI C standard says as little as possible about long int and long double, but there is no reason for us to be similarly reticent. I recommend modifying 13.2:8, rule [2] to read:

> Of sequences not mentioned in [1], those that contain only integral promotions; conversions from float to double, float or double to long double; and trivial conversions are better than all others.

A similar situation exists for long int, but I am a little uncertain how to extend 13.2:8 to cover this without changing the definition of integral promotions to include long types. This may not be a bad idea, but it needs additional investigation.