X3J16/97-0046
WG21/N1084
June 2, 1997
J. Stephen Adamczyk (jsa@edg.com)
John H. Spicer (jhs@edg.com)

The weight of the lvalue-to-rvalue conversion in overload resolution

Introduction

Some aspects of a clarification of overload resolution introduced at the Stockholm meeting (motion 15, N0972=96-0154) cause some behavior that is overly sensitive to the presence or absence of an lvalue-to-rvalue conversion. This new sensitivity breaks an example that appears in the WP, an example from the ARM, and one case from the defining paper on partial ordering of function templates. To fix these problems, the WP should be changed so that the lvalue-to-rvalue conversion is not considered in deciding whether one conversion sequence is a subsequence of another.

The problem

Implicit conversion sequences in overload resolution are described as sequences of standard and user-defined conversions in a canonical order. If the conversion sequence that converts a given argument expression to the corresponding parameter type of function 1 is a subsequence of the conversion sequence needed to convert that same argument expression to the corresponding parameter type of function 2, function 1 is considered better than function 2 on that parameter.

Lvalue-to-rvalue conversions (which include array-to-pointer and function-to-pointer conversions) are one of the steps in a conversion sequence. Therefore, a conversion sequence that is an identity (empty) sequence is a subsequence of one that contains just an lvalue-to-rvalue conversion. This means that an lvalue-to-rvalue conversion can be responsible for selecting one function over another. That seems overly subtle.

It gets even worse when a parameter requiring an lvalue-to-rvalue conversion competes with a parameter of reference type. If the argument expression is an lvalue of the proper type, the reference binds directly to it and there is no associated conversion, i.e., it's an identity conversion sequence. This is considered better than a conversion sequence that consists of just an lvalue-to-rvalue conversion. Therefore “X&” wins over “X” for an lvalue argument of type “X”. This may have some merit for classes, where passing the class by reference is cheaper than passing it by value using a copy constructor, but the distinction seems moot when considering a builtin type.

Examples

From 13.3.3.2 [over.ics.rank] paragraph 3, first bullet, fourth sub-bullet:

```
int g(const int&);
int g(int);
int i;
int k = g(i);       // WP example says this is ambiguous
```

The call in the last line is no longer ambiguous. The implicit conversion sequence for the "`g(const int&)`" function is an identity sequence because the reference binds directly to the lvalue. The implicit conversion sequence for the "`g(int)`" function is an lvalue-to-rvalue conversion sequence. Therefore, the conversion sequence in the first case is a subsequence of the conversion sequence in the second case, and the first function is preferred.

This example can be made ambiguous by changing the argument of the call to an rvalue of the same type:

```
int k = g(1);
```

but it's hard to see why this should be ambiguous when the lvalue case is not.

Another example comes from section 13.4.2 of the ARM, page 333 (thanks to Jason Merrill for pointing this one out):

```
struct X {
  X operator+(int);
};
X operator+(X,double);
void g(X b) {
  X a;
  a = b+1.0;        // ARM says ::operator+(X,double)
}
```

The implicit conversion sequences are as follows:

`X::operator+`
X lvalue --> X& (identity)
double --> int (conversion)

`::operator+`
X lvalue --> X (lvalue-to-rvalue)
double --> double (identity)

So the first argument is better for the member function, but the second argument is better for the non-member function, and the call is ambiguous.

And finally, an example related to partial ordering of function templates:

```
template <class T> struct A { };
template <class T> void f(const T&);
template <class T> void f(A<T>);
void x() {
  A<int> a;
  f(a);
}
```

It was intended that the “`f(A<T>)`” version would be chosen. However, for partial ordering to apply the two functions must be equally good in terms of the conversions involved. Instead, the first function has an identity conversion, and the second an lvalue-to-rvalue conversion, so the first function is chosen and partial ordering of function templates isn’t even considered.

It has been argued that “`f(A<T>)`” is not really a “specialization” of “`f(const T&)`”, because one function takes a reference and the other does not. But the partial ordering rules for function templates, unlike the partial specialization rules for classes, must be able to produce an ordering for unrelated functions produced by different authors. The current WP rules make it impossible to use partial ordering to prefer a function that takes a value parameter — regardless of how closely that parameter type’s structure might mirror the argument type — if there also exists a function with the same name that takes a `T&` parameter.

The suggested fix

The fix is not to consider lvalue-to-rvalue conversions in comparing implicit conversion sequences to see if one is a subsequence of another.

In 13.3.3.2 [over.ics.rank], paragraph 3, first bullet, first sub-bullet, add the indicated text:

- `S1` is a proper subsequence of `S2` (comparing the conversion sequences in the canonical form defined by [over.ics.scs], *excluding any Lvalue Transformation*; the identity conversion sequence is considered to be a subsequence of any non-identity conversion sequence) or, if not that,

In 13.3.3.1 [over.best.ics] paragraph 6, change the following text:

> When the parameter has a class type and the argument expression is an rvalue of the same type, the implicit conversion sequence is an identity conversion. When the parameter has a class type and the argument expression is an lvalue of the same type, the implicit conversion sequence is an lvalue-to-rvalue conversion. When the parameter has a class type and the argument expression is an rvalue of a derived class type, the implicit conversion sequence is a derived-to-base Conversion from the derived class to the base class. [Note: there is no such standard conversion; this derived-to-base Conversion exists only in the description of implicit conversion sequences.] When the parameter has a class type and the argument expression is an lvalue of a derived class type, the implicit conversion sequence is an lvalue-to-rvalue conver-

sion followed by a derived-to-base Conversion. A derived-to-base Conversion has Conversion rank (over.ics.scs).

to the following:

When the parameter has a class type and the argument expression has the same type, the implicit conversion sequence is an identity conversion. When the parameter has a class type and the argument expression has a derived class type, the implicit conversion sequence is a derived-to-base Conversion from the derived class to the base class. [Note: there is no such standard conversion; this derived-to-base Conversion exists only in the description of implicit conversion sequences.] A derived-to-base Conversion has Conversion rank (over.ics.scs).