X3J16/96-0035
WG21/N0853

Eliminating Conversions 'wherever necessary' in Expressions

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

January 30, 1996

Introduction

Paragraph 9 of [expr] in the Working Paper reads as follows:

> User-defined conversions of class types to and from fundamental types, pointers, and so on, can be defined (_class.conv_). If unambiguous (_over.match_), such conversions are applied wherever a class object appears as an operand of an operator or as a function argument (_expr.call_).

Approximately the same thing appears in the ARM, and it was necessary there. In the present-day Working Paper, however, there are detailed descriptions of the circumstances in which specific conversions apply, and this blanket rule is no longer needed. In fact, it's confusing. This paper discusses the remaining cases where the blanket rule might apply, and proposes removal of the rule.

The specific rules

The important addition to the WP in this area is 13.6 [over.built]. That section defines pseudo-prototypes for the built-in operators. For example, for the two-operand operator "`+`" there are pseudo-prototypes for

promoted-arithmetic-type `+` promoted-arithmetic-type
pointer-to-object-type `+` promoted-integral-type
promoted-integral-type `+` pointer-to-object-type

and they compete with any declared `operator+` functions in overload resolution on any "`+`" operation that has at least one operand of class or enumeration type. That gives meaning to an example like

```
struct A {
  operator int();
} a;
main () {
  int i = a + 2.0;
}
```

Specifically, in that case `a` is converted to `int` by calling the conversion function; the result of that conversion is converted to `double`; `2.0` is added; and the result is converted to `int`.

Clause 4 [conv] defines implicit conversion in terms of initialization, and 8.5 [dcl.init] and 8.5.3 [dcl.init.ref] give rules about user-defined conversions in initializations; those affect expression operators defined in terms of initialization, e.g., arguments of a function call and many instances of `static_cast`.

Why get rid of the blanket rule?

The blanket rule makes certain cases arguable, e.g.,

```
struct A {
  operator int *();
} a;
main () {
  delete a;  // Okay?
  reinterpret_cast<char *>(a);  // Okay?
}
```

In cases like this, some people will say "of course it's well-formed; the rule says you can do a conversion wherever you need to," and others will say "no, that's silly; it doesn't mean *that* case."

One can argue about those individual issues, and later in this paper I will make specific recommendations, but the important point is this: if conversions are supposed to apply in a specific case, it's much clearer to say so explicitly. If we want "`delete a`" above to be well-formed, we should indicate what it means in the section on the `delete` operator. Having a blanket "wherever needed" rule is just asking for different people to draw different conclusions about these cases.

So the important part of this proposal is to delete the blanket rule. It's not needed anymore. If certain operators need additional wording to define allowable conversions, we can deal with that on a case-by-case basis.

Easy cases

Of the cases where there's still some question about the allowability of user-defined conversions, most are easy. All the following are silly, in my opinion, and shouldn't allow any special conversions:

```
struct A {
  operator int *();
} a;
struct B {
  int i;
};
struct C {
  operator B();
} c;
struct D {
    operator B*();
  } d;
  main () {
    delete a;                        // error (no conversion to int*)
    reinterpret_cast<char *>(a);  // error (no conversion to int*)
    const_cast<const int *>(a);   // error (no conversion to int*)
    int B::*pm = &B::i;
    c.*pm;                          // error (no conversion to B)
    dynamic_cast<void *>(d);      // error (no conversion to B*)
  }
```

(Note that cfront allows the `delete` case. I do not know if it is widely used, though I suspect not since EDG has never gotten a complaint about that.)

For all these cases, I recommend no change to the WP, and therefore after the removal of the blanket rule these cases would be ill-formed.

The harder case

The only difficult case is the "?" operator.

```
  struct A {
    operator int();
  } a;
  struct B : public A { } b;
  struct C {
    C(int);
    operator int();
  } c(1);
  struct E;
  struct D {
    D();
    D(E);
  } d;
  struct E {
    E();
    E(D);
  } e;
  main () {
    int x = 1;
    x ? a : a;  // Definitely okay (same type)  (1)
    x ? a : b;  // Definitely okay (inheritance-related classes) (2)
    x ? a : x;  // a.operator int() : int    ?? (3)
    x ? c : x;  // c.operator int() : int    ?? (4)
                //                 c : C(int) ??
    x ? d : e;  //              E(d) : e      ?? (5)
                //                 d : D(e)   ??
  }
```

The first two cases are covered by existing wording in 5.16 [expr.cond]. The others, if well-formed, depend on the blanket conversion rule.

Most existing implementations allow case 3, but the consensus breaks down on case 4, where many implementations find the conversion function but not the constructor, one finds (only) the constructor, and one finds both and calls the case ambiguous. Most implementations rejected any case that involved two operands of unrelated class types, like case 5, even if the ambiguity is removed by eliminating the conversion in one direction.

I think that the only reasonable choices are to allow no user-defined conversions (i.e., disallow 3–5), or to consider all possible user-defined conversions (i.e., allow 3, find ambiguity on 4–5). That would rule out the popular existing practice result of accepting case 4 without an ambiguity (because the solution using the constructor is not considered).

Given that, and given the otherwise divergent existing practice, my recommendation is to allow no user-defined conversions in this case, i.e., make no change to the WP. Cases 3–5 would be ill-formed. One can always put in an explicit cast if one wants to write something like that.

If the working group chooses to go in the other direction, I believe that should be accommodated by adding after the following existing wording in 5.16 [expr.cond] paragraph 2

> If the second and the third operands are lvalues and have the same type (before any implicit conversions), the result is an lvalue of that type. Otherwise, if the second and the third operands are lvalues of inheritance-related class types, the operands are implicitly converted to a common type (which shall be a cv-qualified or cv-unqualified version of the type of either the second or third operand) as if by a `static_cast` to a reference to the common type (_expr.static.cast_). [Note: this conversion will be ill-formed if the base class is inaccessible or ambiguous.] The result is an lvalue of the common type.

the following new wording

> Otherwise, if at least one of the second and third expressions has a class type, then if the second expression can be converted to the type of the third expression, the conversion is done and the result has the type of the third expression; if the third expression can be converted to the type of the second expression, the conversion is done and the result has the type of the second expression; if the conversion can be done in both directions, the program is ill-formed.

Working Paper changes

Delete [expr] paragraph 9, which reads

> User-defined conversions of class types to and from fundamental types, pointers, and so on, can be defined (_class.conv_). If unambiguous (_over.match_), such conversions are applied wherever a class object appears as an operand of an operator or as a function argument (_expr.call_).