

Explicit Conversion Operator Draft Working Paper

Lois Goldthwaite, Michael Wong

IBM

michaelw@ca.ibm.com

ACCU

Lois@LoisGoldthwaite.com

Document number: N2223=07-0083

Date: 2007-03-11

Project: Programming Language C++, Evolution Working Group

Reply-to: Michael Wong (michaelw@ca.ibm.com)

Revision: 2

NOTE: This paper supplies the changes to the Working Paper for Lois Goldthwaite's [n1592], Explicit Conversion Operators. This paper was approved in Berlin 2006 and wording was asked for C++0x.

This paper proposes a small change in C++ grammar to permit the function-specifier 'explicit' to be applied to the definition of a user-defined conversion operator. The semantic effect is to inhibit automatic conversions in situations where they may not have been intended.

The Problem

One of the design principles of C++ is that the language does not enforce a different syntax for user-defined types and built-in primitive types. A variable of either category can be passed by value (assuming the programmer has not intentionally disabled this), and a variable of any type can be passed by reference.

The compiler will perform automatic promotions and conversions, if necessary, when numeric types are used as function parameters or when differing types are combined with an operator (int to long, signed to unsigned, float to double, etc.). Similarly, the programmer can write conversion functions for user-defined types, so that the conversions will take place transparently. This is a feature, and A Good Thing, as it decreases the number of overloaded functions which would otherwise be needed [D&E 3.6.1].

In *Modern C++ Design*, Alexandrescu says, "User-defined conversions in C++ have an interesting history. Back in the 1980s, when user-defined conversions were introduced, most programmers considered them a great invention. User-defined conversions promised a more unified type system, expressive semantics, and the ability to define

new types that were indistinguishable from built-in ones. With time, however, user-defined conversions revealed themselves as awkward and potentially dangerous."

The simplest way to perform a user-defined conversion in C++ is with a one-argument constructor in the class of the destination type. However, sometimes the compiler will find and execute a converting constructor in situations not intended by the programmer. Explicit constructors (including copy constructors) were added to prevent unintended conversions being silently called by the compiler, especially when they may be followed by another conversion or promotion. But a constructor is not the only mechanism for transforming one type into another -- the language also allows a class to define a conversion operator, and when called implicitly this may open a similar hole in the type system.

For example, consider a theoretical smart pointer class. The programmer might reasonably want to make it usable in the same contexts as a primitive pointer, and so would want to support a common idiom:

```
template <class T> class Ptr
{
    // stuff
public:
    operator bool() const
    {

        if( rawptr_ )
            return true;
        else
            return false;
    }

private:
    T * rawptr_;
};

Ptr<int> smart_ptr( &some_variable );

if( smart_ptr )
{
    // pointer is valid
}
else
{
    // pointer is invalid
}
```

However, providing such a conversion would also make the compiler uncomplainingly accept code which was semantic nonsense:

```
Ptr<int> p1;
Ptr<float> p2;
std::cout << "p1 + p2 = " << p1 + p2 << std::endl; // prints 0, 1, or 2
```

```
Ptr<Apple> sp1;
Ptr<Orange> sp2; // Orange is unrelated to Apple
if (sp1 == sp2) // Converts both pointers to bool
                // and compares results
```

For this reason, some class authors resort to more complicated constructs to support the idiom without the dangerous consequences:

```
template <class T> class Ptr
{
    // stuff
public:
    struct PointerConversion
    {
        int valid;
    };
    operator int PointerConversion::*() const
    {
        return rawptr_? &PointerConversion::valid : 0;
    }
private:
    T * rawptr_;
};
```

This is now the orthodox technique to support implicit conversion to a bool type, without opening the hole of unintended conversion to arithmetic types. The technique is used in `boost::shared_ptr`, described in [N1450] and the Library TR in progress. But while it serves the purpose, it is difficult for novices to understand why such a circumlocution is necessary, or why it works at all (see the thread at http://www.experts-exchange.com/Programming/Programming_Languages/Cplusplus/Q_20833198.html - registration is encouraged, but not required to read this page).

C++ Templates [C++Templates] by Vandevorde and Josuttis (in B.2.1) gives an example of how user-defined conversion functions can lead to unexpected results (in this case a compiler error):

```
#include <stddef.h>
```

```

class BadString {
public:
BadString( char const * );
// ...

// character access through subscripting:
char& operator[] ( size_t ); // 1
char const& operator[] ( size_t ) const;

// implicit conversion to null-terminated byte string:
operator char* (); // 2
operator char const* ();
// ...
};

int main()
{
BadString str("korrekt");
str[5] = 'c'; // possibly an overload resolution ambiguity!
}

```

Depending on the typedef of `ptrdiff_t`, the compiler may deduce that there is an ambiguity between `BadString::operator[]` and converting the implied "this" argument to `char *` and using the built-in subscript operator. The ambiguity arises on some platforms and not others, which makes the problem even harder for novices to understand. This paper proposes that an 'explicit' function-specifier would make the conversion operator invisible for copy-initialization, in parallel with explicit constructors, which are visible only for direct-initialization.

When conversion is useful, but implicit conversion is dangerous, the recommended approach is to use a named function. One example of this is the `c_str()` member of `std::string`. This approach works well when the destination type is known at compile time, but when templates are involved, it becomes problematic. How can one write generic code for a user-supplied class that may define a function called `to_string()`, or `ToString()`, or `to_String()`, or ...? And when the destination type could be anything, predicting the name becomes impossible.

Even if the committee were to mandate a naming convention for such functions (and I *hate* "standards" based on naming conventions), it would constitute an unwarranted trespass on the programmer's freedom. In contrast, `operator T()` is the accepted way to express such an intent.

```
T t = u.operator T();
```

is straightforward and can be called explicitly when needed. The same applies to

```
T t = static_cast<T>(u) ;
```

Generic programming demands syntactic regularity.

Intended Usage

The intent of this proposal is that explicit-qualified conversion functions would work in the same contexts (direct-initialization, explicit type conversion) as explicit-qualified constructors, and produce diagnostics in the same contexts (copy-initialization) as such constructors do:

```
class U; class V;

class T
{
public:

    T( U const & );
    //implicit converting ctor
    explicit T( V const & );
    // explicit ctor

};

class U
{
};

class V
{
};

class W
{
public:

    operator T() const;
};

class X
{
public:

    explicit operator T() const; // theoretical
};

int main()
{
    U u; V v; W w; X x;
```

```

    // Direct initialization:
    T t1( u );
    T t2( v );
    T t3( w );
    T t4( x );

    // Copy initialization:
    T t5 = u;
    T t6 = v; // error
    T t7 = w;
    T t8 = x; // would be error

    // Cast notation:
    T t9 = (T) u;
    T t10 = (T) v;
    T t11 = (T) w;
    T t12 = (T) x;

    // Static cast:
    T t13 = static_cast<T>( u );
    T t14 = static_cast<T>( v );
    T t15 = static_cast<T>( w );
    T t16 = static_cast<T>( x );

    // Function-style cast:
    T t17 = T( u );
    T t18 = T( v );
    T t19 = T( w );
    T t20 = T( x );

    return 0;
}

```

Why would someone ever choose to write a conversion operator instead of a constructor for the destination type, especially since explicit constructors are already available? One circumstance would be when the programmer does not "own" the destination class -- perhaps it is part of a commercial library, and source code may not even be available. Another circumstance would be when the destination type is a primitive -in which case writing a constructor is not an option.

As is general practice, it is expected that the use of explicit casts would suppress warning messages.

Alternatives

Before adding a new feature to the core language, it is necessary to consider library-based solutions and other alternatives to accomplish the same purpose.

The technique of using named conversion functions is always available, and completely prevents unintentional conversions. But, as mentioned above, it is ill-suited to generic programming.

Some people have proposed an all-purpose templated conversion function:

```
namespace std
{ template< class T, class U > T convert( U const & u ) {
  return T( u ); }
}
```

Since this relies on exactly the missing feature which is proposed here, it would have to be specialized for each pair of types which do not themselves define conversions. In addition to being clumsier to write, it also encounters many of the same issues that bedevil the use of `std::swap()` as a customization point, including the prohibition on partial specialization of function templates.

Another possible path would be via a templated member conversion operator:

```
class M
{
  public:

  template < class To >
  operator To()
  {

  return static_cast<To>( *this );
  }
};

M m;
T t21( m );
```

This would also have to be specialized for each target type. But fatally, it is overly broad. Whether it is specialized or not, it permits conversion to any type which can be used in a single-argument constructor of T (T, U, and V in the above example), thus causing ambiguity.

Proposed Changes to the Working Paper based on [n2135]

Change 7.1.2 [dcl.fct.spec] paragraph 6 as follows:

The explicit specifier shall be used only in declarations of constructors **or conversion functions** within a class declaration; see 12.3.1 and 12.3.2.

Change 12.3 [class.conv] paragraph 1 as follows:

Type conversions of class objects can be specified by constructors and by conversion functions. These conversions are called user-defined conversions and are used for implicit type conversions (clause 4), for initialization (8.5), and for explicit type conversions (5.4, 5.2.9). **User-defined conversions may be implicit, or explicit (7.1.2).**

Change 12.3.2 [class.conv.fct] paragraph 1 as follows:

A member function of a class X with a name of the form

conversion-function-id:

operator conversion-type-id

*function-specifier*_{opt} [*function-specifier-seq*] *conversion-type-id*

Change 12.3.2 [class.conv.fct] paragraph 5 as follows:

Conversion functions are inherited. **An inherited conversion function which is declared explicit in the base class is explicit in the derived class.**

Add 12.3.2 [class.conv.fct] paragraph 6 as follows:

An explicit conversion function constructs objects just like non-explicit conversion functions, but does so only where the direct-initialization syntax (8.5) or where casts (5.2.9, 5.4) are explicitly used. [Example:

```
class T { };

class X
{
public:

explicit operator T() const;
};

int main()
{
X x;

// Direct initialization:
T t4( x );

// Copy initialization:
T t8 = x; // error

// Cast notation:
T t12 = (T) x;
```



```
// Static_cast:  
T t16 = static_cast<T>( x );  
  
// Function-style cast:  
T t20 = T( x );  
  
return 0;  
}
```

-- end example]

Add Appendix A [gram] A.6 as follows after *function-specifier*:

function-specifier-seq:
*function-specifier-seq*_{opt} *functionl-specifier*

Reference

[n1592] Explicit Conversion Operator, <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1592.pdf>

[n1450] A Proposal to Add General Purpose Smart Pointers to the Library Technical Report, <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1450.html>

[n2135] Programming Languages —C++

[C++Templates] C++ Templates: The Complete Guide, Daveed Vandevoorde, Nicolai M. Josuttis. Addison Wesley, 2002.

[D&E] Design and Evolution of C++, Bjarne Stroustrup, Addison Wesley, 1998.