Document Number: SC22/WG21/N1592=04-0032
Date: 2004-2-13
Author: Lois Goldthwaite
Email: Lois@LoisGoldthwaite.com

Explicit Conversion Operators

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. 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 by Vandevoorde 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("correkt");
      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 less preferred when such an ambiguity arises. (If the explicit conversion operator were the best match, it should produce a diagnostic.)

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;

    // New 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 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), causing ambiguity.

Changes to the standard

12.3.2 is the section of the standard which defines class member conversion operators. The grammar would be expanded to allow qualified declarations:

conversion-function-id:
operator conversion-type-id
function-specifier $_{opt}$ *[sequence of function specifiers] conversion-type-id*

The [sequence of function specifiers] is to allow more than one of "inline," "virtual," and "explicit" to be applied to the same conversion operator (12.3.2p6 already says that conversion functions can be virtual).

12.3.2p5 says that conversion functions are inherited. Add: An inherited conversion function which is declared explicit in the base class is explicit in the derived class as well.

7.1.2p6 The explicit specifier shall be used only in declarations of constructors [add: or conversion operators] within a class declaration; see class.conv.ctor [add: and class.conv.fct].

Wordsmithing by a core expert is cordially invited. (I assume "virtual" and "inline" function-specifiers aren't mentioned in 12.3.2 because they can be applied to conversion operators under the grammar found in some other clause. 9.3.1p4 says "A non-static member function may be declared virtual or pure virtual." 7.1.2 discusses inline functions. Also see core issue 194.)

Other considerations not addressed

In C++, if both the source and destination classes define a conversion facility, the compiler finds the ambiguity and gives a diagnostic. Nothing in this proposal would change that rule.

C++ currently requires that conversion functions be non-static member functions of either the source type or the destination type. This paper does not propose any change to that requirement. Free-standing or static member conversion functions are not proposed in this paper.

In a series of conversions, only one user-defined conversion sequence is allowed, possibly combined with one or more standard conversions. Nothing in this proposal would change that rule.

A common use of a conversion operator is to support the "if(smart_ptr)" idiom mentioned earlier in the paper. Should an explicit conversion operator be called here, implicitly? My inclination is to say no -- the purpose of explicit conversions is to favor safety over convenience. Some instructors already teach their pupils to write out all boolean expressions in full ("if(ptr.get() != 0)"), and languages like Java and C# do not have implicit conversions even from numeric types to a boolean expression, so it is feasible to write programs without using the idiom.

A special exception could be argued that if() and while() blocks always need a bool as the controlling expression, and therefore should be considered explicit calls to the conversion. However, a simple cast or a self-documenting function would be enough to invoke the conversion only in desired circumstances:

```
template <class T>
 bool is_valid( T const & something )
{
   return static_cast<bool>( something );
}

if( is_valid( smart_ptr ) ) ...
```

For people who prefer the purity of the classic idiom, the somewhat obscure implicit conversion to pointer-to-member is still available for use.

Summary

Explicit conversion operators would correct an asymmetry in the present language, in which converting constructors defined by the destination class can be qualified as explicit, but conversion operators in the source class cannot. [Strictly speaking, explicit constructors are not classed as converting constructors. See 12.3.1. But I'm not sure how else to describe a one-argument constructor which is not copying another object of the same type.] Applying the same syntax and semantics to both contexts should make C++ easier to learn and teach.

A desire for explicit-qualified conversion operators is a recurring theme on the newsgroups. My colleagues who teach introductory C++ say their students often question the lack of this facility in C++, especially if they have experience with C#. (C# allows conversion operators to be qualified with keywords "implicit" or "explicit." Explicit conversions can only be performed by using cast syntax.)

Being able to rely on regular syntax (function-style casts or static_casts) to convert between types, without the risk of unintentional conversions, would provide better support for generic programming.