Explicit Conversion Operator Draft Working Paper (revision 2)

Lois Goldthwaite, Michael Wong, Jens Mauer, Alisdair Meredith
Lois@LoisGoldthwaite.com
michaelw@ca.ibm.com
Jens.Maurer@gmx.net
public@alisdairm.net

Document number: N2380=07-0240
Date: 2007-08-01
Project: Programming Language C++, Core Working Group
Reply-to: Michael Wong (michaelw@ca.ibm.com)
Revision: 2

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.

This paper introduces a new term "implicitly boolean-converted" for use in contexts where a boolean value is expected (but not an integral value). This term is linked to direct-initialization, and overload resolution is amended to invoke explicit conversion functions for direct-initialization only.

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

History

Previous papers: N1592, Explicit Conversion Operators, was approved in Berlin 2006 and wording was asked for C++0x. N2223 was an attempt to provide that wording; this paper presents further revised syntax drafted by Jens Maurer.

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](N1450) and ISO/IEC TR 19768, C++ Library Extensions, and since incorporated into the C++0x working paper at 20.6.6.2.5. 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 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.

But conversions to boolean or pointer types are not the only use cases for explicit conversion operators. Robert Klarer has stated that the decimal floating point library described in N2198 (or WDTR 24733) would benefit from extra programmer control over the conversion of decimal floating point values to native integral or binary floating point values. This is what motivated the evolution working group to approve N1592 in Berlin.

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.

Primitive destination types raise special syntactic problems, not only because they do not have constructors but because direct initialization and copy initialization have the same semantics, as explained in 8.5.

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.

Proposed Changes to the Working Paper based on [n2315]

Paper N2223 "Explicit Conversion Operator Draft Working Paper" by Lois Goldthwaite and Michael Wong presents suggested changes to the Working Paper N2315 to support explicit conversion operators. The previous version of that paper was N1592 "Explicit Conversion Operators" by Lois Goldthwaite.

This paper introduces a new term "implicitly boolean-converted" for use in contexts where a boolean value is expected (but not an integral value). This term is linked to direct-initialization, and overload resolution is amended to invoke explicit conversion functions for direct-initialization only.

Proposed wording

Modify clause 4 conv paragraph 3 as indicated:

> An expression e can be *implicitly converted* to a type T if and only if the declaration `T t=e;` is well-formed, for some invented temporary variable t (8.5). <u>An expression e can be *implicitly boolean-converted* if and only if the declaration `bool t(e);` is well-formed, for some invented temporary variable t (8.5 dcl.init).</u> The effect of ~~the~~ <u>either</u> implicit conversion is the same as performing the declaration and initialization and then using the temporary variable as the result of the conversion. The result is an lvalue if T is an lvalue reference type (8.3.2), and an rvalue otherwise. The expression e is used as an lvalue if and only if the initialization uses it as an lvalue.

Modify 5.3.1 expr.unary.op paragraph 8 as indicated:

> The operand of the logical negation operator ! is implicitly <u>boolean-</u>converted ~~to bool~~ (clause 4 conv); its value is true if the converted operand is false and false otherwise. The type of the result is bool.

Modify 5.14 expr.log.and paragraph 1 as indicated:

> The && operator groups left-to-right. The operands are both implicitly boolean-converted ~~to type bool~~ (clause 4 conv). The result is true if both operands are true and false otherwise. Unlike &, && guarantees left-to-right evaluation: the second operand is not evaluated if the first operand is false.

Modify 5.15 expr.log.or paragraph 1 as indicated:

> The || operator groups left-to-right. The operands are both implicitly boolean-converted ~~to bool~~ (clause 4 conv). It returns true if either of its operands is true, and false otherwise. Unlike |, || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to true.

Modify 5.16 expr.cond paragraph 1 as indicated:

> Conditional expressions group right-to-left. The first expression is implicitly boolean-converted ~~to bool~~ (clause 4). It is evaluated and if it is true, the result of the conditional expression is the value of the second expression, otherwise that of the third expression. ...

Modify 5.19 expr.const paragraph 4 (modified by N2235) as indicated:

> If an expression of literal class type is used in a context where an integral constant expression is required, then that class type shall have a single non-explicit conversion function to an integral or enumeration type and that conversion function shall be constexpr. ...

Modify 6.4 stmt.select paragraph 4 as indicated:

> The value of a condition that is an initialized declaration in a statement other than a `switch` statement is the value of the declared variable implicitly boolean-converted ~~to type bool~~. If that conversion is ill-formed, the program is ill-formed. The value of a condition that is an initialized declaration in a `switch` statement is the value of the declared variable if it has integral or enumeration type, or of that variable implicitly converted to integral or enumeration type otherwise. The value of a condition that is an expression is the value of the expression, implicitly boolean-converted ~~to bool~~ for statements other than `switch`; if that conversion is ill-formed, the program is ill-formed. The value of the condition will be referred to as simply "the condition" where the usage is unambiguous.

Modify 7 dcl.dcl paragraph 4 as indicated:

> In a *static_assert-declaration* the *constant-expression* shall be ~~an integral~~ a constant expression (5.19 expr.const) that can be implicitly boolean-converted (clause 4). If the value of the expression when ~~converted to bool~~ so converted is true, the declaration has no effect. ...

Modify 7.1.2 dcl.fct.spec paragraph 6 as indicated:

> The explicit specifier shall be used only in the declaration of a constructor or conversion function within its class definition; see 12.3.1 class.conv.ctor and 12.3.2 class.conv.fct.

Modify 8.5 dcl.init paragraph 11 as indicated:

> The form of initialization (using parentheses or =) is generally insignificant, but does matter when the initializer or the entity being initialized has a class type; see below. A parenthesized initializer can be a list of expressions only when the entity being initialized has a class type.

Modify 12.3.2 class.conv.fct paragraph 2 as indicated:

A conversion function may be explicit (7.1.2 dcl.fct.spec), in which case it is only considered as a user-defined conversion for direct-initialization (8.5 dcl.init). Otherwise, user-defined ~~User-defined~~ conversions are not restricted to use in assignments and initializations. [Example:

```
    class Y { };

    struct Z {
      explicit operator Y() const;
      // ...
    };

    void h(Z z) {
      Y y1(z);     // ok: direct-initialization
      Y y2 = z;    // ill-formed: copy-initialization
      Y y3 = (Y)z; // ok: cast notation
    }

    void g(X a, X b)
    {
        int i = (a) ? 1+a : 0;
        int j = (a&&b) ? a+b : i;
        if (a) {                // ...
        }
    }
```

-- end example]

Modify 13.3.1.1.2 over.call.object paragraph 2 as indicated (add "non-explicit" twice):

In addition, for each non-explicit conversion function declared in T of the form

```
       operator conversion-type-id ( ) cv-qualifier ;
```

where *cv-qualifier* is the same cv-qualification as, or a greater cv-qualification than, *cv*, and where *conversion-type-id* denotes the type "pointer to function of (P1,...,Pn) returning R", or the type "reference to pointer to function of (P1,...,Pn) returning R", or the type "reference to function of (P1,...,Pn) returning R", a surrogate call function with the unique name *call-function* and having the form

```
       R call-function ( conversion-type-id F, P1 a1,
...   ,Pn an) { return F (a1,... ,an); }
```

is also considered as a candidate function. Similarly, surrogate call functions are added to the set of candidate functions for each non-explicit conversion function declared in a base class of T provided the function is not hidden within T by another intervening declaration [Footnote: ...].

Modify 13.3.1.4 over.match.copy paragraph 1, second bullet as indicated (add "non-explicit" once):

When the type of the initializer expression is a class type "*cv* S", the non-explicit conversion functions of S and its base classes are considered. Those that are not hidden within S and yield a type whose cv-unqualified version is the same type as

T or is a derived class thereof are candidate functions. Conversion functions that return "reference to X" return lvalues or rvalues, depending on the type of reference, of type X and are therefore considered to yield X for this process of selecting candidate functions.

Modify 13.3.1.5 over.match.conv paragraph 1, first bullet as indicated:

> The conversion functions of S and its base classes are considered, except that for copy-initialization other than in a context where an implicit boolean-conversion (clause 4) applies, only the non-explicit conversion functions are considered. Those non-explicit conversion functions that are not hidden within S and yield type T or a type that can be converted to type T via a standard conversion sequence (13.3.3.1.1) are candidate functions. Those explicit conversion functions that are not hidden within S and yield type T or a type that can be converted to type T with a qualification conversion (4.4 conv.qual) are also candidate functions. Conversion functions that return a cv-qualified type are considered to yield the cv-unqualified version of that type for this process of selecting candidate functions. Conversion functions that return "reference to cv2 X" return lvalues or rvalues, depending on the type of reference, of type cv2 X" and are therefore considered to yield X for this process of selecting candidate functions.

Editing note: It is not permitted to initialize a double via an explicit conversion function that returns a float or an int. The proposed text also prevents slicing by inadvertant derived-to-base value conversions.

Modify 13.3.1.6 over.match.ref paragraph 1, first bullet as indicated:

> The conversion functions of S and its base classes are considered, except that for copy-initialization, only the non-explicit conversion functions are considered. Those that are not hidden within S and yield type "lvalue reference to cv2 T2", where "cv1 T" is reference-compatible (8.5.3 dcl.init.ref) with "cv2 T2", are candidate functions.

Editing note: The only implicit conversions possible here are the reference-to-derived-to-reference-to-base and qualification conversions. Neither should be restricted when an explicit conversion function is selected.

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

[n2315] 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.