A Proposal to Tweak Certain C++ Contextual Conversions, v2

Document #: WG21/N3306 = PL22.16/11-0076
Date: 2011-09-02
Revises: N3253
Project: Programming Language C++
Reply to: Walter E. Brown<wb@fnal.gov>
FPE Dept., Computing Division
Fermi National Accelerator Laboratory
Batavia, IL 60510-0500

Contents

1 Introduction

The context in which a C++ expression appears often influences how the expression is evaluated, and therefore may impose requirements on the expression to ensure such evaluation is possible. For example, it is well-understood that an expression used in an **`if`**, **`while`**, or similar context must be convertible to **`bool`** so that the expression can be converted to **`bool`** during its evaluation. This conversion is termed *contextual*, and is set forth in [conv]/3.

In four cases, the FDIS (N3290) uses different language to specify an analogous context-dependent conversion. In those four contexts, when an operand is of class type, that type must have a "single non-explicit conversion function" to a suitable (context-specific) type. Here are the relevant excerpts (bold emphasis added), in order of occurrence:

- [expr.new]/6: "The *expression* in a *noptr-new-declarator* shall be of integral type, unscoped enumeration type, or a **class type for which a single non-explicit conversion function** to integral or unscoped enumeration type exists (12.3)."
- [expr.delete]/1: "The operand shall have a pointer to object type, or a **class type having a single non-explicit conversion function** (12.3.2) to a pointer to object type."
- [expr.const]/5: "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`**."
- [stmt.switch]/2: "The condition shall be of integral type, enumeration type, or of a **class type for which a single non-explicit conversion function** to integral or enumeration type exists (12.3)."

Each excerpt cited above is followed by language stating or implying that the required conversion function will be used to convert an operand of class type and that the resulting converted value will be used instead of the original. However, consider the small but very reasonable examples presented in the next section.

2 Motivating example

Listing 1 presents a modest template that wraps a value of arithmetic or pointer type **T**, ensuring that the wrapped value will by default be initialized with **T**'s zero value. Note in particular the pair of conversion operators (lines 13-14), provided according to a well-established C++ coding pattern that allows **const** and non-**const** objects to be treated differently:

```
1  //                               Listing 1
2  template< class T
3          , class = typename std::enable_if<  std::is_arithmetic<T>::value
4                                           || std::is_pointer   <T>::value
5                                           >::type
6          >
7  class zero_init
8  {
9  public:
10   zero_init( )       : val( static_cast<T>(0) )  { }
11   zero_init( T val ) : val( val )                { }

13   operator T & ( )        { return val; }
14   operator T   ( ) const  { return val; }

16 private:
17   T val;
18 };
```

Listing 2 shows a simple use of this template. We observe (verified with two modern compilers) that most expressions (*e.g.*, ***p**) have no compilation issue, but that the expression **delete p** produces diagnostics. More embarrassingly, the equivalent expressions **delete (p+0)** and **delete +p** are fine:

```
//                               Listing 2
zero_init<int*> p;  assert( p == 0 );
p = new int(7);     assert( *p == 7 );
delete p;       // error!
delete (p+0);   // okay
delete +p;      // also okay
```

Listing 3 shows another use of this template. Here, too, the arithmetic, assignment, and comparison operations work fine, yet the simple expression **i** is embarrassingly ill-formed (although the equivalent **i+0** and **+i** are well-formed!) when used as the **switch** condition:

```
//                               Listing 3
zero_init<int> i;  assert( i == 0 );
i = 7;             assert( i == 7 );
switch( i   ) { ... }  // error!
switch( i+0 ) { ... }  // okay
switch( +i  ) { ... }  // also okay
```

3 Discussion

The principal issue, in each of the four contexts cited in the Introduction, seems to lie in their common helpful but very strict requirement that limits a class to only one conversion operator while allowing for the conversion of a value of the class's type to a corresponding value of a type specified by the context. As shown earlier, such a limitation disallows straightforward code; the above examples must work around the restriction by otherwise uselessly adding zero or unnecessarily applying unary **+** (so that the conversion occurs in a different context that obeys different rules). We conclude from this that the current "single non-explicit conversion function" approach is unnecessarily strict, as it is less helpful than it could be and thus forces workarounds that seem silly.

Another concern is the scope of the qualifier "single" in the current wording. Must there be but a single conversion function in the class, or may there be several so long as a single one is appropriate to the context? The current language seems unclear on this point.

It is also unclear whether a conversion operator that produces a reference to an appropriate type is an appropriate conversion operator. (A question on this point was posted to the Core reflector on 2011-02-21, but has gone unanswered as of this writing.) Current compiler practice seems to admit such operators, but the current language seems not to.

Just as it does not allow a class to treat **`const`** and non-**`const`** objects of its type differently, the current approach also seems not always to admit a conversion function that may be declared in a base class. The concern here is the slight dichotomy in the current wording: Two cases speak of the class "having" an appropriate conversion function, while the other two require that the function "exist". If the conversion function is in a base class, it "exists", but the class at issue is not the one that "has" the function.

To address all these concerns, we recommend instead to use the proven approach typified by the term *contextually converted to* ***`bool`*** as defined in [conv]/3. We therefore propose a modest addition to [conv]/3 to define contextual conversion to other specified types, and then appeal to this new definition in rewording the four contexts cited in the Introduction.

Not only does the proposed change avoid the need for the embarrassing workarounds shown above, it brings the benefit of more consistent behavior by eliminating four special cases for contextual conversions, aligning them with the rules for the C++11 contextual conversion to **`bool`**.

4 Proposed wording

The wording proposed in this section is based on that in FDIS N3290, with deleted text shown ~~thus~~ and new language like this. The first proposed change is to define *contextually implicitly converted* as a new term of art; the remaining changes then use this term in rewording the four paragraphs called out above.

4.1 Changes to [conv]

3 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).

— Certain language constructs require that an expression be converted to a Boolean value. An expression **`e`** appearing in such a context is said to be *contextually converted to* ***`bool`*** and is well-formed if and only if the declaration **`bool t(e);`** is well-formed, for some invented temporary variable **`t`** (8.5).

— Certain other language constructs require similar conversion, but to a value having one of a specified set of types appropriate to the construct. An expression **`e`** of class type **`E`** appearing in such a context is said to be *contextually implicitly converted* to a specified type and is well-formed if and only if the declaration **`T t=e;`** is well-formed, for some invented temporary variable **`t`** (8.5) of a type **`T`** determined as follows: **`E`** is searched for an overload set of conversion operators whose return type is *cv* **`T`** or reference to *cv* **`T`** such that **`T`** is allowed by the context. There shall be exactly one such **`T`**.

The effect of ~~either~~ any implicit conversion is the same as performing the corresponding 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 or an rvalue reference to function type (8.3.2), an xvalue if **`T`** is an rvalue reference to object type, and a prvalue otherwise. The expression **`e`** is used as a glvalue if and only if the initialization uses it as a glvalue.

Drafting Notes:

- The above wording introduces a short bullet list to clarify the text's linguistic and technical structure by visually associating related sentences. Although we rather like the appearance of the resulting exposition, the list format is an optional part of the proposal.
- We considered appending a Note to the first bullet to draw attention that it admits **`explicit`** conversion operators, but decided that would exceed the scope of this proposal. However, it is an open question whether the new second bullet should similarly admit **`explicit`** conversion operators; see *Option 2*, below.
- One reviewer proposed *contextually copy-converted* as the new term of art. While we wouldn't object to this change, we believe this term stresses the proposed mechanism rather than the intent, and so mildly prefer *contextually implicitly converted* as proposed herein.

4.2 Changes to [expr.new]

6 Every *constant-expression* in a *noptr-new-declarator* shall be an integral constant expression (5.19) and evaluate to a strictly positive value. The *expression* in a *noptr-new-declarator* shall be of integral type, unscoped enumeration type, or ~~a~~ class type ~~for which a single non-explicit conversion function to integral or unscoped enumeration type exists (12.3)~~. If the expression is of class type, the expression is ~~converted by calling that conversion function, and the result of the conversion is used in place of the original expression~~ contextually implicitly converted (4.1) to an integral or unscoped enumeration type. [*Example:* . . . — *end example*]

4.3 Changes to [expr.delete]

1 The *delete-expression* operator destroys a most derived object (1.8) or array created by a *new-expression*. . . . . The operand shall ~~have a~~ be of pointer to object type~~,~~ or ~~a~~ class type ~~having a single non-explicit conversion function (12.3.2)~~. If the operand is of class type, the operand is contextually implicitly converted (4.1) to a pointer to object type. The *delete-expression*'s result has type **`void`**.[78]

4.4 Changes to [expr.const]

5 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~~ expression is contextually implicitly converted (4.1) to an integral or unscoped enumeration type and ~~that~~ the applicable conversion function shall be **`constexpr`**. [*Example:* . . . — *end example*]

4.5 Changes to [stmt.switch]

2 The condition shall be of integral type, enumeration type, or ~~of a~~ class type ~~for which a single non-explicit conversion function to integral or enumeration type exists (12.3)~~. If the condition is of class type, the condition is contextually implicitly converted ~~by calling that conversion function and the result of the conversion is used in place of the original condition for the remainder of this section~~ (4.1) to an integral or enumeration type. Integral promotions are performed. Any statement within the **switch** statement can be labeled with one or more case labels as follows: . . . .

5 Option 1

This section proposes alternate wording to [expr.new]/6. The first proposed change appeals in part to a different term of art, *converted constant expression*; the remaining two changes are consequential adjustments presented for the sake of consistency.

5.1 Changes to [expr.new]

6 Every *constant-expression* in a *noptr-new-declarator* shall be ~~an integral constant expression (5.19)~~ a converted constant expression (5.19) of type **std::size_t** and evaluate to a strictly positive value. The *expression* in a *noptr-new-declarator* ~~shall be of integral type, unscoped enumeration type, or a class type for which a single non-explicit conversion function to integral or unscoped enumeration type exists (12.3). If the expression is of class type, the expression is converted by calling that conversion function, and the result of the conversion is used in place of the original expression~~ is implicitly converted to **std::size_t**. [*Example:* . . . — *end example*]

5.2 Changes to [expr.const]

3 . . . . [*Note:* such expressions may be used in **new** expressions (5.3.4), as case expressions (6.4.2), as enumerator initializers if the underlying type is fixed (7.2), as array bounds (8.3.4), and as integral or enumeration non-type template arguments (14.3). — *end note*] . . . .

5.3 Changes to [dcl.array]

1 . . . . If the *constant-expression* (5.19) is present, it shall be ~~an integral constant expression~~ a converted constant expression (5.19) of type **std::size_t** and its value shall be greater than zero. . . . .

6 Option 2

This section addresses the issue of copy- *vs.* direct-initialization in the proposed wording. It's unclear that the affected contexts should refrain from considering explicit conversions when analogous conversions to **bool** do not so refrain. Surely **switch**(. . .), for example, is no less explicit a context than **if**(. . .), **while**(. . .), *etc.* The necessary adjustment to the above-proposed wording is tiny, but the entire proposed paragraph is presented here to provide context.

6.1 Changes to [conv]

3 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).

— Certain language constructs require that an expression be converted to a Boolean value. An expression **e** appearing in such a context is said to be *contextually converted to* ***bool*** and is well-formed if and only if the declaration **bool t(e);** is well-formed, for some invented temporary variable **t** (8.5).

— Certain other language constructs require similar conversion, but to a value having one of a specified set of types appropriate to the construct. An expression **e** of class type **E** appearing

> in such a context is said to be *contextually converted* to a specified type and is well-formed if and only if the declaration `T t(e);` is well-formed, for some invented temporary variable `t` (8.5) of a type `T` determined as follows: `E` is searched for an overload set of conversion operators whose return type is *cv* `T` or reference to *cv* `T` such that `T` is allowed by the context. There shall be exactly one such `T`.

The effect of ~~either~~ any implicit conversion is the same as performing the corresponding 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 or an rvalue reference to function type (8.3.2), an xvalue if `T` is an rvalue reference to object type, and a prvalue otherwise. The expression `e` is used as a glvalue if and only if the initialization uses it as a glvalue.

6.2 Consequential adjustments

Since "implicitly" is removed from the term of art under this Option 2, it must also be removed where the term is applied. The changes to [expr.new], [expr.delete], and [stmt.switch] proposed in §4 are thusly affected. The same is true for the proposed changes to [expr.const] unless Option 1 (§5), which employs a different term of art, is elected.

7 Acknowledgments

Many thanks to the reviewers of early drafts of this paper for their helpful and constructive comments, and especially to James Widman for his very able drafting assistance and to Jens Maurer for the idea underlying Option 1. We also acknowledge the Fermi National Accelerator Laboratory's Computing Division, sponsor of our participation in the C++ standards effort, for its past and continuing support of our efforts to improve C++ for all our user communities.

8 Changes from N3253

1. Apply editorial tweaks.
2. Refer to FDIS N3290 as the base document rather than to Working Draft N3225, and adjust the cited and proposed text accordingly.
3. Exhibit and mention unary **+** as a second weird workaround for the status quo.
4. Introduce, discuss, and apply *contextually implicitly converted* as the new term of art, replacing the simpler but potentially ambiguous *contextually converted*.
5. Tighten the language regarding the type `T` to be used in contextual implicit conversion, and adjust accordingly the sections applying the revised language.
6. Introduce as Option 1 alternate wording for [expr.new].
7. Raise the issue of admitting `explicit` conversion operators, and introduce Option 2 for consideration.
8. Update the acknowledgments.