Doc. no.: N2687=08-0197
Date: 2008-06-25
Project: Programming Language C++
Reply to: Alberto Ganesh Barbati
<ganesh@barbati.net>

Forward declaration of enumerations (rev. 2)

Revision history

N2687	rev. 2	Changes in the grammar proposed in Sophia: *enum-specifier* reverted to require an *enum-body* and a new *enum-declaration* non-terminal added to *block-declaration*. Minor editorial changes.
N2568	rev. 1	Incorporates comments from the EWG that lead to minor changes in the proposed wording of [basic.def]/2 and [dcl.type.elab]/3. Moreover, a new informative section has been added to provide more context and rationale about the proposed changes to [dcl.enum]/7.
N2499		Initial revision

1 Introduction

In C++03 every declaration of an enumeration is also a definition and must include the full list of enumerators. The list is always needed to determine the underlying type of the enumeration, which is necessary to generate code that manipulates values of the enumerations. However, there are use cases where it would be desirable to declare an enumeration without providing the enumerators. The compiler could still generate meaningful code, if at least the underlying type is known. The syntax introduced by paper N2347[1)], which allow the programmer to explicitly specify the underlying type, can easily be extended to cover this scenario.

2 Motivation

2.1 Reduce coupling

Consider the header file of a component providing support for localized strings:

```
// file locstring.h

#include <string>

enum localized_string_id
{
    /* very long list of ids */
};

std::istream& operator>>(std::istream& is, localized_string_id& id);

std::string get_localized_string(localized_string_id id);
```

The enumeration `localized_string_id` may have several hundreds entries and be generated automatically by a tool, rather than manually maintained; changes can therefore be very frequent. Every component that needs a localized string will eventually need to include `locstring.h` and therefore will have to be recompiled every time the enumeration changes.

[1)] paper N2347 has been integrated in the draft for C++0X, that is paper N2606 at the time of writing.

Now, consider the following piece of code:

```
localized_string_id id;
std::cin >> id;
std::cout << get_localized_string(id);
```

Does this code depend on the list of enumerators? According to C++03 the answer is yes, because C++03 requires the presence of the entire list of enumerators to determine the underlying type of `localized_string_id`. Of course, if we didn't know the underlying type, we couldn't instantiate the variable `id` nor we pass it by value to function `get_localized_string()`. However, neither the names nor the values of the enumerators are actually used by the code! If we could just tell the compiler the underlying type, there would be no technical obstacle for it to produce the correct code even in absence of the list of enumerators.

2.2 Type-safe data hiding

Consider this class:

```
class C
{
public:
    /* public interface */

private:
    enum E { /* enumerators */ };
    E e;
};
```

According to C++03, the list of enumerators is required in order to determine the underlying type of the data member `e`, an essential information needed to determine the layout of class `C`. If the public interface of class `C` does not make any use of `E`, the list of enumerators of `E` would be merely an implementation detail, yet any change to the list requires re-compilation of all clients of class `C`.

Moreover, if class `C` is part of the interface of a closed-source library which is distributed in binary form, the names of the enumerator may need to be obfuscated in order to avoid disclosing internal details.

The obvious work-around is to define the member variable `e` with a basic integral type and then declare the enumeration in another file. This approach is inferior, because we lose the type safety provided by the enumeration.

3 Proposal

This proposal introduces a syntax that allows declaring an enumeration without providing a list of enumerators. Such declaration would not be definition in order to avoid problems with ODR and can be provided only for enumerations with fixed underlying type. An enumeration can be then be redeclared, possibly providing the missing list of enumerators, but the redeclaration shall match the previous declaration:

```
enum E : short;         // OK: unscoped, underlying type is short
enum F;                 // illegal:  enum-base is required
enum class G : short;   // OK: scoped, underlying type is short
enum class H;           // OK: scoped, underlying type is int

enum E : short;         // OK: redeclaration of E
enum class G : short;   // OK: redeclaration of G
enum class H;           // OK: redeclaration of H
enum class H : int;     // OK: redeclaration of H
```

```
enum class E : short;   // illegal:  previously declared as unscoped
enum G : short;         // illegal:  previously declared as scoped

enum E;                 // illegal:  enum-base is required
enum E : int;           // illegal:  different underlying type
enum class G;           // illegal:  different underlying type
enum class H : short;   // illegal:  different underlying type

enum class H { /* */ }; // OK: this redeclaration is a definition
```

The underlying type must be specified each time as a mean to avoid possibile interpretation ambiguities that could depend on the order of the declarations.

Moreover, enumerations declared at class or namespace scope can be defined in an enclosing scope:

```
struct S
{
    enum E : int;       // unscoped enumeration, underlying type is int
    E e;                // e is an implemented as-if it was declared int
};

enum S::E : int         // definition of the nested enumeration
{
  /* ... */
};
```

4 Interaction with N2347

N2347 changed the definition of *elaborated-type-specifier* by allowing *enum-key*s where only the `enum` keyword was previously allowed. Moreover, "the *enum-key* used in an *elaborated-type-specifier* need not match the one in the enumeration's definition." The author of this proposal believes this change was both unnecessary and a mistake. Moreover, it is an impediment for this proposal so a return to the previous definition is deemed necessary.

Paper N2347 makes this code legal:

```
enum class E { a, b };
enum E x = E::a;        // OK
```

however, it also makes this code legal:

```
enum E { a, b };
enum class E x = a;     // OK ???
```

which doesn't look as good as in the previous case: the extra `class` keyword in the second line is confusing to say the least. The objections to the change in the definition of *elaborated-type-specifier* can be summarized as follows:

a) *elaborated-type-specifier*s are used mainly for compatibility with legacy C code, they are not needed in practice in C++, where `E` is just as good as `enum E`, but it's shorter. Legacy C code won't need to support scoped enumerations explicitly

b) if `enum E` can be used in place of `enum class E`, the programmers will probably prefer the former, especially since adding `class` is not a reliable source of additional information about `E`

c) allowing `enum class E` to refer to an unscoped enumeration can be a source of confusion

d) the change was inessential to the other important changes introduced by paper N2347

The conflict with this proposal arises when parsing this declaration:

```
enum class E;           // (1)
```

With N2347 wording, such code is ill-formed because an *elaborated-type-specifier* is the "sole constituent" of the declaration and the form is not explicitly listed as legal in [dcl.type.elab]/1. We can't just add (1) to the list of legal forms, because `E` may still refer to either a scoped or unscoped enumeration and this makes a lot of difference, because scoped enumerations always have fixed underlying type while unscoped enumerations don't.

According to this proposal, line (1) would unambigously declare `E` as a scoped enumeration with underlying type of `int`. The `enum` keyword would still be allowed to refer to scoped enumerations, while `enum class` and `enum struct` would be banned from *elaborated-type-specifier*s, for example:

```
enum class E { a, b };
enum E x1 = E::a;       // OK in N2347, OK in this proposal
enum class E x2 = E::a; // OK in N2347, illegal in this proposal

enum F { a, b };
enum F y1 = a;          // OK in N2347, OK in this proposal
enum class F y2 = a;    // OK in N2347, illegal in this proposal
```

Notice that the following:

```
enum E;                 // illegal
```

would remain illegal (as it is in both C++03 and N2347), because an *elaborated-type-specifier* is the "sole constituent" of a declaration and this form is not among the allowed forms in [dcl.type.elab]/1. Instead, none of the following

```
enum E : int;           // OK: E is unscoped, underlying type is int
enum class F;           // OK: F is scoped, underlying type is int
```

would trigger [dcl.type.elab]/1, because in the first case the *elaborated-type-specifier* is no longer the "sole constituent" of the declaration, while in the second case there is no *elaborated-type-specifier*.

5 Impact on the standard and implementability

This proposal provides a semantic to a syntax that was previously illegal and does not change the semantic of code that was legal in C++03. The new syntax does not introduce new keywords. Code that was legal according to N2347, however, can become illegal.

There are no known or anticipated difficulties in implementing these features.

6 Proposed text

In this section, changes are presented as modifications to existing wording in current draft, paper N2606, where ~~strikethrough text~~ refers to existing text that is to be deleted, and underscored text refers to new text that is to be added.

6.1 Changes to clause 3.1 [basic.def]

Changes to paragraph 2:

> A declaration is a *definition* unless it declares a function without specifying the function's body (8.4), it contains the `extern` specifier (7.1.1) or a *linkage-specification* (7.5) and neither an *initializer* nor a *function-body*, it declares a static data member in a class definition (9.4), it

is a class name declaration (9.1), it is an *enum-declaration* (7.2), or it is a typedef declaration (7.1.3), a *using-directive* (7.3.3), or a *using-declaration* (7.3.4).

6.2 Changes to clause 3.3.1 [basic.scope.pdecl]

Changes to paragraph 3:

The point of declaration for a class first declared by a *class-specifier* is immediately after the identifier or *simple-template-id* (if any) in its *class-head* (clause 9). The point of declaration for an enumeration is immediately after the identifier (if any) in its *enum-specifier* (7.2) or after its first *enum-declaration* (7.2), whichever comes first." The point of declaration of a template alias immediately follows the identifier for the alias being declared.

6.3 Changes to clause 7 [dcl.dcl]

Changes to paragraph 1:

block-declaration:
simple-declaration
asm-definition
namespace-alias-definition
using-declaration
using-directive
static_assert-declaration
alias-declaration
enum-declaration

6.4 Changes to clause 7.1.6.3 [dcl.type.elab]

Changes to this clause effectively restore the C++03 wording. Only the example introduced by N2347 is retained, as it is still valid.

elaborated-type-specifier:
class-key $::_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
class-key $::_{opt}$ *nested-name-specifier*$_{opt}$ `template`$_{opt}$ *simple-template-id*
~~*enum-key*~~`enum` $::_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*

Changes to paragraph 3:

The *class-key* or ~~*enum-key*~~`enum` keyword present in the *elaborated-type-specifier* shall agree in kind with the declaration to which the name in the *elaborated-type-specifier* refers. This rule also applies to the form of *elaborated-type-specifier* that declares a *class-name* or `friend` class since it can be construed as referring to the definition of the class. Thus, in any *elaborated-type-specifier*, the ~~*enum-key*~~`enum` keyword shall be used to refer to an enumeration (7.2), the `union` *class-key* shall be used to refer to a union (clause 9), and either the `class` or `struct` *class-key* shall be used to refer to a class (clause 9) declared using the `class` or `struct` *class-key*. ~~The *enum-key* used in an *elaborated-type-specifier* need not match the one in the enumeration's definition.~~ [*Example:*

```
enum class E { a, b };
enum E x = E::a; // OK
```

— *end example*]

6.5 Changes to clause 7.2 [dcl.enum]

Changes to paragraph 1:

enum-name:
identifier
enum-specifier:
~~*enum-key identifier*$_{opt}$ *enum-base*$_{opt}$ { *enumerator-list*$_{opt}$ }~~
~~*enum-key identifier*$_{opt}$ *enum-base*$_{opt}$ { *enumerator-list* , }~~
enum-head { *enumerator-list*$_{opt}$ }
enum-head { *enumerator-list* , }
enum-head:
enum-key identifier$_{opt}$ *enum-base*$_{opt}$
enum-key nested-name-specifier identifier enum-base$_{opt}$
enum-declaration:
enum-key identifier enum-base$_{opt}$;
enum-key:
`enum`
`enum class`
`enum struct`
enum-base:
: *type-specifier-seq*
enumerator-list:
enumerator-definition
enumerator-list , *enumerator-definition*
enumerator-definition:
enumerator
enumerator = *constant-expression*
enumerator:
identifier

Changes to paragraph 2:

> The enumeration type declared with an *enum-key* of only `enum` is an *unscoped enumeration*, and its *enumerators* are *unscoped enumerators*. The *enum-keys* `enum class` and `enum struct` are semantically equivalent; an enumeration type declared with one of these is a *scoped enumeration*, and its *enumerators* are *scoped enumerators*. The *type-specifier-seq* of an *enum-base* shall name an integral type; any cv-qualification is ignored. An *enum-declaration* declaring an unscoped enumeration shall not omit the *enum-base*. [...]

Add as a new paragraph after paragraph 3:

> An *enum-declaration* is is either a redeclaration of an enumeration in the current scope or a declaration of a new enumeration. [*Note:* an enumeration declared by an *enum-declaration* has fixed underlying type and is a complete type. The list of enumerators can be provided in a later redeclaration with an *enum-specifier*. —*end note*] A scoped enumeration shall not be later redeclared as unscoped or with a different underlying type. An unscoped enumeration shall not be later redeclared as scoped and each redeclaration shall include an *enum-base* specifying the same underlying type.

Add as a new paragraph, either after the previously added one or at the end of the clause, at the project editor's discretion:

> If the *enum-key* is followed by a *nested-name-specifier*, the *enum-specifier* shall refer to an enumeration that was previously declared directly in the class or namespace to which the *nested-name-specifier* refers (i.e., neither inherited nor introduced by a *using-declaration*), and the *enum-specifier* shall appear in a namespace enclosing the previous declaration.

7 About the redeclaration syntax

The proposed wording in [dcl.enum]/7 requires the programmer to specify the underlying type on each redeclaration including the definition, as in:

```
enum E : int;
enum E : int { a, b, c };
```

This redundant syntax may seem verbose and error prone. Two alternatives were considered in a discussion on comp.std.c++, namely:

a) a redeclaration of a previously declared enumeration shall not specify an enum-base, the underlying type is the one determined by the first declaration.

b) a redeclaration of a previously declared enumeration shall either have no enum-base or have an enum-base specifying the same type as the underlying type of the first declaration. In any case, the underlying type is the one determined by the first declaration.

The problem with these approaches is that they allow the presence or absence of a previous declaration to change the meaning of an otherwise perfectly valid definition:

```
// E.h

enum E       { ea, eb }; // underlying type of E is implementation-defined
enum class F { fa, fb }; // underlying type of F is int

// A.cpp

enum E : short;
enum class F : short;

#include "E.h" // defines both E and F with underlying type short
```

This can be more disorientating than requiring the programmer to be verbose. In a certain sense, the proposed wording is actually less error prone than the alternatives, as mistakes can be easily detected and correctly diagnosed by compiler.

8 Acknowledgments

The author is grateful to Lawrence Crowl and Jens Maurer for their feedback.