

Doc No: SC22/WG21/N1579
J16/04-0019

Date: February 10, 2004

Project: JTC1.22.32

Reply to: Herb Sutter
Microsoft Corp.
1 Microsoft Way
Redmond WA USA 98052
Email: hsutter@microsoft.com

David E. Miller
Atlantic International Inc.
67 Wall Street, 22nd floor
New York NY 10005
Email: j16p0403@atl-intl.com

Strongly Typed Enums

1. Overview	2
2. The Problem, and Current Workarounds.....	3
2.1. Problem 1: Implicit conversion to an integer	3
2.2. Problem 2: Inability to specify underlying type	4
2.2.1. Predictable and specifiable space	4
2.2.2. Predictable/specifiable type (notably signedness)	4
2.3. Problem 3: Scope	5
2.4. Problem 4: Incompatible extensions to address these issues.....	6
3. Proposal	6
3.1. Create a new kind of enum that is strongly typed: enum class	7
3.2. Extend existing enums: Underlying type and explicit scoping.....	8
3.3. One more extension to existing enums: No implicit conversions.....	9
4. Interactions and Implementability	10
4.1. Interactions	10
4.2. Implementability	10
5. Proposed Wording	10
5.1. Main change: Updating [dcl.enum] and the Annex A grammar.....	11
5.2. Other core changes	15
5.3. Recommended library changes.....	16
6. References	17

1. Overview

“C enumerations constitute a curiously half-baked concept.”

— [Stroustrup94], p. 253

C++ [C++03] currently provides only incremental improvements over C [C99] enums. Major safety and security problems remain, notably in the areas of type safety, unintended errors, code clarity, and code portability. Worse, in practice these problems typically manifest as *silent* behavioral changes when programs are compiled using different compilers, including different versions of the same compiler. The results from such silent safety holes can be catastrophic, particularly in life-critical software, and we should therefore close as many as we can.

Today’s workarounds boil down to not using enums, or at least never exposing them directly. Some of the workarounds require heroic efforts on the part of library authors and/or users to provide what should be a basic and safe feature.

This paper proposes extensions to enums that will reduce the likelihood of undetected errors while enabling code to be written more clearly and portably. The proposed changes are pure extensions to ISO C++ that will not affect the meaning of existing programs.

This paper is a revision of [Miller03] incorporating direction from the Evolution Working Group at the October 2003 WG21 meeting. In particular, the EWG direction was that the proposal should be revised to:

- focus on three specific problems with C++ enums (their implicit conversion to integer, the inability to specify the underlying type, and the absence of strong scoping);
- come up with a different syntax than originally proposed;
- provide a distinct new enum type having all the features that are considered desirable; and
- provide pure backward-compatible extensions for existing enums with a subset of those features (e.g., the ability to specify the underlying type).

The proposed syntax and wording for the distinct new enum type is based on the C++/CLI [C++/CLI-WD1.1] draft syntax for this feature. The proposed syntax for extensions to existing enums is designed for similarity.

This proposal falls into the following categories:

- Improve support for library building and security, by providing better type safety without manual workarounds.
- Make C++ easier to teach and learn, by removing common stumbling blocks that trip new programmers.
- Improve support for systems programming, particularly for programmers targeting platforms such as [CLI] that already provide native support for strongly typed enums.

- Remove embarrassments. People with a vested interest in promoting other languages love this kind of situation because it lets them buttress claims that C++ is “too hard” and “not type-safe.” We should take away the ammunition.

2. The Problem, and Current Workarounds

2.1. Problem 1: Implicit conversion to an integer

Current C++ enums are not type-safe. They do have some type safety features; in particular, it is not permitted to directly assign from one enumeration type to another, and there is no implicit conversion from an integer value to an enumeration type. But other type safety holes exist notably because “[t]he value of an enumerator or an object of an enumeration type is converted to an integer by integral promotion” ([C++03] §7.2(8)).

For example:

```
enum Color { ClrRed, ClrOrange, ClrYellow, ClrGreen, ClrBlue, ClrViolet };
enum Alert { CndGreen, CndYellow, CndRed };

Color c = ClrRed;
Alert a = CndGreen;

a = c;                                // error
a = ClrYellow;                         // error
bool armWeapons = ( a >= ClrYellow );  // ok; oops
```

The current workaround is simply not to use the enum. At minimum, the programmer manually wraps the enum inside a class to get type-safety:

```
class Color {                                // class simplified for clarity
    enum Color_ { Red_, Orange_, Yellow_, Green_, Blue_, Violet_ };
    Color_ value;

public:
    static const Color Red, Orange, Yellow, Green, Blue, Violet;

    explicit Color( Color& other )           : value( other.value ) { }

    bool operator<( Color const& other )     { return value < other.value; }

    int toInt() const                       { return value; }
};

const Color Color::Red( Color::Red_ );
// etc.

// ... here, repeat all the above scaffolding for Alert ...
```

```
Alert a = Alert::Green;  
bool armWeapons = ( a >= Color::Yellow );    // error
```

2.2. Problem 2: Inability to specify underlying type

Current C++ enums have an implementation-defined underlying type, and this type cannot be specified explicitly by the programmer. This causes two related problems that merit distinct attention.

2.2.1. Predictable and specifiable space

It can be necessary to specify definitely how much space will be used by the representation of an enumeration variable, particularly to be able to lay out fields in a struct with the expectation those fields will have the same sizes and layouts across multiple compilers, as in data communications and storage applications. Because current C++ enums allow implementations to take either the minimal space necessary or a larger amount, they cannot be used reliably in such structures.

For example, consider the following subtle portability pitfall:

```
enum Version { Ver1 = 1, Ver2 = 2 };  
  
struct Packet {  
    Version ver;                                // bad, size can vary by implementation  
    // ... more data ...  
  
    Version getVersion() const { return ver; }  
};
```

The current workaround is, again, not to use the enum:

```
enum Version { Ver1 = 1, Ver2 = 2 };  
  
struct Packet {  
    unsigned char ver;                            // works, but requires casting  
    // ... more data ...  
  
    Version getVersion() const { return (Version)ver; }  
};
```

2.2.2. Predictable/specifiable type (notably signedness)

It can be necessary to specify how a value of the enumeration will be treated when used as a number, notably whether it will be signed or unsigned. The difference can affect program correctness, and it should be able to make this portably reliable without requiring heroic effort on the part of the library writer or user.

For example, consider the behavior of enum `E` in this code, where the naïve user declared `Ebig` using a constant ending in a suffix specifying unsignedness and expected the compiler to understand the intent:

```
enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFFF0U };

int main() {
    cout << sizeof( E ) << endl;
    cout << "Ebig = " << Ebig << endl;
    cout << "E1 ? -1 =\t" << ( E1 < -1 ? "less" : E1 > -1 ? "greater" : "equal" ) << endl;
    cout << "Ebig ? -1 =\t" << ( Ebig < -1 ? "less" : Ebig > -1 ? "greater" : "equal" ) << endl;
}
```

This result of all three tests (the value of `Ebig`, and `E1`'s and `Ebig`'s comparisons to `-1`) is actually implementation-defined and thus nonportable. This is counter-intuitive to users.

To illustrate, here is a sampling of the variety of results across compilers on the same Windows XP test platform, all of which report `sizeof( E )` to be 4:

Compiler	Ebig = ?	E1 ? -1	Ebig ? -1	Warning
Borland 5.5.1	-16	greater	less	<i>none</i>
Digital Mars 8.38	4294967280	greater	greater	<i>none</i>
Comeau 4.3.3 (EDG 3.3)	4294967280	less	less	integer conversion resulted in a change of sign
gcc 2.95.3	4294967280	less	less	comparison between signed and unsigned
gcc 3.3.2	4294967280	less	less	comparison between signed and unsigned integer expressions
Metrowerks CodeWarrior 8.3	-16	greater	less	<i>none</i>
Microsoft Visual C++ 6.0	-16	greater	less	<i>none</i>
Microsoft Visual C++ 7.1	4294967280	less	less	<i>none</i>
Microsoft Visual C++ 8.0 (alpha)	-16	greater	less	signed/unsigned mismatch

Note the variance of behaviors across compilers, and from version to version of the same compiler.

It would be better if it were possible to easily write more portable code. Current workarounds require forgoing enums and instead writing class wrappers (as in §2.1) or explicit casts (as in §2.2.1).

2.3. Problem 3: Scope

Current C++ enums are not strongly scoped. In particular:

- It is not legal for two enumerations in the same scope to have enumerators with the same name. For example:

```
enum E1 { Red };  
enum E2 { Red };    // error
```

- More generally, the name of an enumerator exists in the enclosing scope, which causes name conflicts and/or surprising results even when the enumerations are in different scopes. For example:

```
namespace NS1 {  
    enum Color { Red, Orange, Yellow, Green, Blue, Violet };  
};  
  
namespace NS2 {  
    enum Alert { Green, Yellow, Red };  
};  
  
using namespace NS1;  
  
NS2::Alert a = NS2::Green;  
bool armWeapons = ( a >= Yellow );    // ok; oops
```

The current workaround is not to use the enum and instead write a class wrapper (as in §2.1).

2.4. Problem 4: Incompatible extensions to address these issues

Implementations already vary widely in practice in some of these areas, as shown in §2.2.2.

Some implementations already have added incompatible extensions to address some of these problems, which is undesirable. It would be better if the extensions were instead provided consistently and reliably as standardized extensions in ISO C++ itself.

3. Proposal

This proposal is in two parts, following the EWG direction to date:

- provide a distinct new enum type having all the features that are considered desirable; and
- provide pure backward-compatible extensions for existing enums with a subset of those features (e.g., the ability to specify the underlying type).

The proposed syntax and wording for the distinct new enum type is based on the C++/CLI [C++/CLI-WD1.1] draft syntax for this feature. The proposed syntax for extensions to existing enums is designed for similarity.

3.1. Create a new kind of enum that is strongly typed: **enum class**

We propose adding a distinct new enum type with the following features:

- *Declaration:* The new enum type is declared using **enum class**, which does not conflict with existing enums and conveys the strongly-typed and strongly-scoped nature of these enums. The body between the braces is the same as for existing enums. For example:

```
enum class E { E1, E2, E3 = 100, E4 /* = 101 */};
```

- *Conversions:* There is no implicit conversion to or from an integer. For example:

```
enum class E { E1, E2, E3 = 100, E4 /* = 101 */};
```

```
void f( E e ) {  
    if( e >= 100 )    // error  
    ;  
}
```

- *Underlying type:* The underlying type is always well-specified. The default is **int**, and can be explicitly specified by the programmer by writing **: type** following the enumeration name, where the underlying type **type** may be any integer or floating point types except **wchar_t**, and the enumeration and all enumerators have the specified type. This underlying type specifier is not required on forward declarations. For example:

```
enum class E : unsigned long { E1 = 1, E2 = 2, Ebig = 0xFFFFFFFF0U };
```

- *Scoping:* Like a class, the new enum type introduces its own scope. The names of enumerators are in the enum's scope, and are not injected into the enclosing scope. For example:

```
enum class E { E1, E2, E3 = 100, E4 /* = 101 */};
```

```
E e1 = E1;    // error  
E e2 = E::E2; // ok
```

The following example, demonstrate how the removal of the implicit conversion and the addition of strong scoping help solve the problems described in §2.1 and §2.3:

```
// no need to prefix the enumerators with "Clr" and "Cnd"  
// because they are not in the same scope  
enum class Color { Red, Orange, Yellow, Green, Blue, Violet };  
enum class Alert { Green, Yellow, Red };
```

```
Color c = Color::Red;    // explicit qualification is required  
Alert a = Color::Green;
```

```
bool armWeapons = ( a >= Color::Yellow );    // error
```

The following example demonstrates how the specification of underlying type helps solve the problem described in §2.2:

```
enum class Version : UINT8 { Ver1 = 1, Ver2 = 2 };

struct Packet {
    Version ver;                // ok, portable (for suitable definitions of UINT8)
    // ... more data ...

    Version getVersion() const { return ver; }
};
```

3.2. Extend existing enums: Underlying type and explicit scoping

We propose extending existing enums with a subset of the features listed in §3.1:

- *Underlying type*: The underlying type may be specified. The default is to follow the existing implementation-defined rules, otherwise the underlying type can be explicitly specified by the programmer by writing *: type* following the enumeration name, where the underlying type *type* may be any integer or floating point types except `wchar_t`, and the enumeration and all enumerators have the specified type. For example:

```
enum E : unsigned long { E1 = 1, E2 = 2, Ebig = 0xFFFFFFFF0U };
```

- *Scoping*: Existing enums now introduce their own scopes. The names of enumerators are in the enum's scope, and they are also injected into the enclosing scope. This design achieves two goals: a) to preserve backward compatibility so that the meaning of existing programs is unchanged; and b) to enable programmers to write enum-agnostic code that operates on both kinds of enums, because enumerators may be (redundantly) referred to by explicit scope qualification using the enum name. For example:

```
enum E { E1, E2, E3 = 100, E4 /* = 101 */ };

E e1 = E1;           // ok
E e2 = E::E2;        // ok
```

The following example demonstrates how the specification of underlying type helps solve the problem described in §2.2:

```
enum Version : UINT8 { Ver1 = 1, Ver2 = 2 };

struct Packet {
    Version ver;                // ok, portable (for suitable definitions of UINT8)
    // ... more data ...

    Version getVersion() const { return ver; }
};
```

3.3. One more extension to existing enums: No implicit conversions

There is one other extension to existing enums which deserves separate consideration because it is the only area where the same feature is exposed using different syntax for existing and the proposed new enums (it is implicit for `enum class` enums):

- *Conversions:* By default there is still an implicit conversion to an integer. By specifying the keyword `explicit`, the implicit conversion to integer is disabled. For example:

```
explicit enum class E { E1, E2, E3 = 100, E4 /* = 101 */ };

void f( E e ) {
    if( e >= 100 )    // error
    ;
}
```

The primary argument in favor of extending existing enums with an optional `explicit` is that it can encourage adoption of this safety feature in two common scenarios. Note in particular that many life-critical systems fall into one or both of these categories:

- *Code bases that are required to be compatible with multiple compilers, including older compilers.* This extension enables such backward compatibility, for example by code such as the following which can be controlled using a compiler command line macro setting, an easy practice during standardized, multi-platform builds:

```
#if( compiler_supports_explicit_enums )
    #define explicit_enum explicit enum
#else
    #define explicit_enum enum
#endif

explicit_enum E { E1, E2, E3 = 100, E4 /* = 101 */ };
```

- *Stabilized code bases that are not open to routine maintenance.* In these cases, even a small change to the code base commonly requires management approval.

This extension would enable a developer to assert to management that the impact is a purely one-word change that can only expose bugs; that is, the argument would be that all that is needed is to make a one-word change to an existing enum declaration and that there will be no other effect on the existing code base except that dubious uses, which will often be bugs, will turn into compiler errors.

The same effect cannot (easily) be achieved by converting existing enums to enum classes, which also enforce strong scoping and so will affect other code besides code that relies on the implicit conversion. The closest approximation might be to convert the existing enum to an enum class accompanied by adding one constant variable for each enumerator having the same name as the enumerator, so that existing code that relies on the name being available will find the constant now that the enumerator is not in the enclosing scope. This is not very maintainable because of the duplication of enumerators with constants and having to change the two in sync, but it could be an acceptable and automatable solution for a one-time lint-like check of the code to expose the places where it relies on the implicit conversion.

It could be argued that this is just one of dozens or hundreds of places where we have unspecified behavior in the standard. We separate this point so that it can be decided separately from the main proposal whether migrating this one case by itself justifies adding a single-purpose extension to the language.

The proposed wording includes this **explicit** feature. To remove this feature from the proposed wording requires only removing “**explicit**_{opt}” from the *enum-class-key* production (two places).

4. Interactions and Implementability

4.1. Interactions

Particularly in the Conversions clause, references to enumerations need to reflect that only non-explicit enumerations have an implicit conversion to an integral type.

The standard allows enumerations and enumerators in places where integral and/or integral constant expressions are allowed or required (e.g., in **switch** statements). Since this does not apply to enumerations that have a floating point underlying type, these places need to be updated to refer to *integer* enumerations (or equivalently *integral* enumerations), enumerations having an integral underlying type.

Generally, places that refer to “integer/integral or enumeration type” need to be updated to “integer/integral or *integer/integral* enumeration type.” Places that refer to “arithmetic or enumeration type” need no change.

By design, there are no effects on legacy code.

4.2. Implementability

There are no known or anticipated difficulties in implementing these features. These features have been implemented in Microsoft Visual C++ 8.0 (alpha).

5. Proposed Wording

In this section, where changes are either specified by presenting changes to existing wording, ~~strike-through text~~ refers to existing text that is to be deleted, and underscored text refers to new text that is to be added.

5.1. Main change: Updating [dcl.enum] and the Annex A grammar

In §A.6, change the production for *enum-specifier* as follows:

enum-specifier:

~~enum~~ enum-class-key *identifier*_{opt} enum-base_{opt} { *enumerator-list*_{opt} }

enum-class-key:

explicit_{opt} enum

enum class

enum struct

enum-base:

: simple-type-specifier

Change §7.2 as follows. Existing footnotes are unchanged, and some existing references to grammar elements have been italicized for consistency (these changes to italics only are unmarked):

7.2 Enumeration declarations

[dcl.enum]

- 1 An enumeration is a distinct type (3.9.1) with named constants. Its name becomes an *enum-name*, within its scope.

enum-name:

identifier

enum-specifier:

~~enum~~ enum-class-key *identifier*_{opt} enum-base_{opt} { *enumerator-list*_{opt} }

enum-class-key:

explicit_{opt} enum

enum class

enum struct

enum-base:

: simple-type-specifier

enumerator-list:

enumerator-definition

enumerator-list , *enumerator-definition*

enumerator-definition:

enumerator

enumerator = *constant-expression*

enumerator:

identifier

An enumeration type declared with an *enum-class-key* of only **enum** is a POE (“plain old enum”), and its *enumerators* are POE *enumerators*. The *simple-type-specifier* of an *enum-base* shall be a fundamental type that is not void or wchar_t. The identifiers in an *enumerator-list* are de-

clared as constants, and can appear wherever constants are required. An *enumerator-definition* with = gives the associated *enumerator* the value indicated by the *constant-expression*. The *constant-expression* shall be of integral, floating point, or enumeration type. If the first *enumerator* has no *initializer*, the value of the corresponding constant is zero. An *enumerator-definition* without an *initializer* gives the *enumerator* the value obtained by increasing the value of the previous *enumerator* by one.

2 [Example:

```
enum { a, b, c=0 };
enum { d, e, f=e+2 };
```

defines a, c, and d to be zero, b and e to be 1, and f to be 3. —end example]

3 The point of declaration for an *enumerator* is immediately after its *enumerator-definition*. [Example:

```
const int x = 12;
{ enum { x = x }; }
```

Here, the *enumerator* x is initialized with the value of the constant x, namely 12. —end example]

4 Each enumeration defines a type that is different from all other types. Each enumeration also has an underlying type; the value of sizeof() applied to an enumeration type, an object of enumeration type, or an enumerator, is the value of sizeof() applied to the underlying type. The underlying type can be explicitly declared using enum-base; if not explicitly declared, the underlying type of a non-POE enumeration type defaults to int. The type of the enumeration has the same representation (including size, bit layout, and alignment requirements) as the underlying type. Following the closing brace of an *enum-specifier*, each *enumerator* has the type of its enumeration. Prior to the closing brace, if the enumeration is a non-POE type or the underlying type is specified, then the type of each enumerator is that of the underlying type; otherwise, the type of each enumerator is the type of its initializing value;

- If an initializer is specified for an *enumerator*, the initializing value has the same type as the expression.
- If no initializer is specified for the first *enumerator*, the ~~type is~~ initializing value has an unspecified integral type.
- Otherwise the type of the initializing value is the same as the type of the initializing value of the preceding *enumerator* unless the incremented value is not representable in that type, in which case the type is an unspecified integral type sufficient to contain the incremented value.

An integer enumeration (or POE) type or integral enumeration (or POE) type is an enumeration (or POE) type having an integral underlying type. A floating point enumeration (or POE) type is an enumeration (or POE) type having a floating point underlying type.

- 5 For a POE type whose underlying type is not explicitly specified, the underlying type ~~The underlying type of an enumeration is a floating point type that can represent all the floating point enumerator values defined in the enumeration (if any of the enumerator values are floating point values) or otherwise an integral type that can represent all the enumerator values defined in the enumeration. It is implementation-defined which integral type is used as the underlying type for an enumeration except that if the underlying type is a floating point type then the underlying shall not be larger than float unless the values of a floating point enumerator cannot fit in or be exactly represented by a float, and if the underlying type is an integral type then the underlying type shall not be larger than int unless the value of an enumerator cannot fit in an int or unsigned int. If the enumerator-list is empty, the underlying type is as if the enumeration had a single enumerator with value 0. The value of sizeof() applied to an enumeration type, an object of enumeration type, or an enumerator, is the value of sizeof() applied to the underlying type.~~
- 6 If the enum-base is specified, then the values of the enumeration are the values of the underlying type specified in the enum-base. ~~Otherwise, f~~For an enumeration where e_{min} is the smallest enumerator and e_{max} is the largest, the values of the enumeration are the values of the underlying type in the range b_{min} to b_{max} , where b_{min} and b_{max} are, respectively, the smallest and largest values of the smallest bit-field that can store e_{min} and e_{max} .⁸¹) It is possible to define an enumeration that has values not defined by any of its enumerators.
- 7 Two enumeration types are layout-compatible if they have the same *underlying type*.
- 8 The value of an *enumerator* or an object of an POE enumeration type is converted to an integer by integral promotion (4.5). [Example:

```
enum color { red, yellow, green=20, blue };
color col = red;
color* cp = &col;
if (*cp == blue)           // ...
```

makes `color` a type describing various colors, and then declares `col` as an object of that type, and `cp` as a pointer to an object of that type. The possible values of an object of type `color` are `red`, `yellow`, `green`, `blue`; these values can be converted to the integral values 0, 1, 20, and 21. Since enumerations are distinct types, objects of type `color` can be assigned only values of type `color`.

```
color c = 1;                // error: type mismatch,
                           // no conversion from int to color

int i = yellow;             // OK: yellow converted to integral value 1
                           // integral promotion
```

—end example]

- 9 An expression of arithmetic or enumeration type can be converted to an enumeration type explicitly. The value is unchanged if it is in the range of enumeration values of the enumeration type; otherwise the resulting enumeration value is unspecified.
- 10 The *enum-name* ~~and each enumerator declared by an *enum-specifier*~~ is declared in the scope that immediately contains the *enum-specifier*. Each *enumerator* is declared at the end its *enumerator-definition* in the scope of its *enum-specifier*, and can be used anywhere without access restriction. If its *enum-specifier* is a POE type, each enumerator is additionally inserted into the scope that immediately contains the *enum-specifier*. These names obey the scope rules defined for all names in (3.3) and (3.4). [Example:

```
enum direction { left='l', right='r' };

void g()
{
    direction d;           // OK
    d = left;              // OK
    d = direction::right;  // OK
    // ...
}
```

—end example]

An *enumerator* declared in class scope can be referred to using the class member access operators (::, . (dot) and -> (arrow)), see 5.2.5. [Example:

```
class X {
public:
    enum direction { left='l', right='r' };
    int f(int i)
        { return i==left ? 0 : i==right ? 1 : 2; }
};

void g(X* p)
{
    direction d;           // error: direction not in scope
    int i;
    i = p->f(left);         // error: left not in scope
    i = p->f(X::right);     // OK
    i = p->f(p->left);       // OK
    // ...
}
```

—end example]

5.2. Other core changes

Change §4.5(2) as follows:

An rvalue of type `wchar_t` (3.9.1) or an integral POE-~~enumeration~~ type (7.2) can be converted to an rvalue of the first of the following types that can represent all the values of its underlying type: `int`, `unsigned int`, `long`, or `unsigned long`.

In §4.6, after paragraph 1 insert the following new paragraph:

An rvalue of floating point POE type (7.2) can be converted to an rvalue of the first of the following types that can represent all the values of its underlying type: `float`, `double`, or `long double`.

In §4.7(1), change “enumeration type” to “integral POE type”.

In §4.9(1), change “floating point type” to “floating point or floating point POE type”.

In §4.9(2), change “enumeration type” to “integer POE type”.

In §4.12(1) and §5(9), change “enumeration type” to “POE type”.

In §5(9) footnote 54, change “enumerated type” to “POE type”.

In §5.2.2(7), change “or a floating point type that is subject to the floating point promotion” to “or a floating point or floating point POE type that is subject to the floating point promotion”.

Change §5.2.9(7) as follows:

A value of integral, floating point or enumeration type can be explicitly converted to an enumeration type. The value is unchanged if the original value is within the range of the enumeration values and representable by the enumeration’s underlying type (7.2). Otherwise, the resulting enumeration value is unspecified.

In §5.2.10(5) and §5.3.1(9), change “enumeration type” to “integral enumeration type”.

In §5.3.4(6), change “non-negative value” to “non-negative integral value”.

In §5.6(2), §5.7(1), §5.7(2) and §5.8(1), change “enumeration type” to “integral enumeration type”.

In §5.11(1), §5.12(1), §5.13(1) and §5.19(1) (six places), change “integral or enumeration” to “integral or integral enumeration”.

In §5.19(1), change “enumerators” to “integral enumerators”.

In §6.4(4) (two places), change “integral or enumeration” to “integral or integral enumeration”.

In §6.4.2(2) (two places) and §7.1.5.1(2), change “enumeration type” to “integral enumeration type”.

In §7.1.5.3 and §A.6, change the production of *elaborated-type-specifier* as follows (note that this also fixes several occurrences of keywords that were not put in code font):

elaborated-type-specifier:

class-key *::opt nested-name-specifier*_{opt} *identifier*

class-key *::opt nested-name-specifier*_{opt} *template*_{opt} *template-id*

~~*enum*~~ *enum-class-key* *::opt nested-name-specifier*_{opt} *identifier*

~~*typename*~~ *typename* *::opt nested-name-specifier* *identifier*

~~*typename*~~ *typename* *::opt nested-name-specifier* ~~*template*~~ *template*_{opt} *template-id*

In §7.1.5.3(3) (two places), change “enum keyword” to “enum-class-key”.

In §9.1(2), change “enumerator” to “POE enumerator”.

In §9.2(1), change “enumerators” to “POE enumerators”, and change “enumeration” to “integral enumeration”.

In §9.2(4), change “enumeration” to “integral enumeration”.

In §9.2(13), change “an enumerated type” to “a POE type”.

In §9.3.1(2), change “an enumerator” to “a POE enumerator”.

In §9.4(3), change “enumerator” to “POE enumerator”.

In §9.4.2(4) and §9.6(3), change “enumeration” to “integral enumeration”.

In §9.7(1) and §9.8(1), change “enumerators” to “POE enumerators”.

In §10.2(5), change “an enumerator” to “a POE enumerator”.

In §13.6(2), change “enumeration” to “POE”.

In §14(5), change “enumerator” to “POE enumerator”.

In §14.1(4), §14.3.2(1), §14.3.2(5), §14.4(1), and §14.6.2.3(2), change “enumeration” to “integral enumeration”.

5.3. Recommended library changes

In §18.2.1.3, change the declaration of `float_round_style` to add the *enum-base* : signed char .

In §18.2.1.4, change the declaration of `float_denorm_style` to add the *enum-base* : signed char .

In §22.2.1, change the declaration of `ctype_base` to add the *enum-base* : unsigned char .

In §22.2.1.5, change the declaration of `codecvt_base` to add the *enum-base* : unsigned char .

In §22.2.5.1, change the declaration of `time_base` to add the *enum-base* : unsigned char .

In §22.2.6.3, change the declaration of `money_base::part` to add the *enum-base* : unsigned char .

In §27.4.2, change the declaration of `ios_base::event` to add the *enum-base* : unsigned char .

In §27.4.2(1), change “an enumerated type, `seekdir`” to “an enumerated type with underlying type unsigned char, `seekdir`”.

6. References

- [C99] *Programming Language C* (ISO/IEC 9899:1999(E)).
- [C++03] *Programming Language C++* (ISO/IEC 14882:2003(E)).
- [C++/CLI-
WD1.1] *C++/CLI Language Specification, Working Draft 1.1, Jan. 2004* (Ecma/TC39-TG5/2004/3).
- [CLI] *Common Language Infrastructure (CLI)* (ECMA-335, 2nd edition, December 2002).
- [Miller03] D. Miller. "Improving Enumeration Types" (ISO/IEC JTC1/SC22/WG21 N1513 = ANSI/INCITS J16 03-0096).
- [Stroustrup94] B. Stroustrup. *The Design and Evolution of C++* (Addison-Wesley, 1994).