Doc No: SC22/WG21/N1719
J16/04-0159

Date: October 21, 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 (revision 1)

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 we should enable making this portably reliable without heroic efforts from 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 = 0xFFFFFFF0U };

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	*none*
Digital Mars 8.38	4294967280	greater	greater	*none*
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	*none*
Microsoft Visual C++ 6.0	-16	greater	less	*none*
Microsoft Visual C++ 7.1	4294967280	less	less	*none*
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 type except wchar_t, and the enumeration and all enumerators have the specified type. For example:

```
enum class E : unsigned long { E1 = 1, E2 = 2, Ebig = 0xFFFFFFF0U };
```

- *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 type except `wchar_t`, and the enumeration and all enumerators have the specified type. For example:

  ```
  enum E : unsigned long { E1 = 1, E2 = 2, Ebig = 0xFFFFFFF0U };
  ```

- *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; }
};
```

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.

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 (beta).

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 <u>underscored text</u> refers to new text that is to be added.

5.1. Updating [dcl.enum]

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-key identifier*$_{opt}$ *enum-base*$_{opt}$ { *enumerator-list*$_{opt}$ }

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

2 An enumeration type declared with an *enum-key* of only `enum` is a *POE* ("plain old enum"), and its *enumerator*s are *POE enumerators*. The *type-specifier-seq* of an *enum-base* shall be a cv-unqualified integral type. The identifiers in an *enumerator-list* are declared 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 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.

[*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 }; }
```

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

5 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 un-

derlying 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.

6 For a POE type whose underlying type is not explicitly specified, the underlying type ~~The *underlying type* of an enumeration~~ is an integral type that can represent all the enumerator values defined in the enumeration. If no integral type can represent all the enumerator values, the enumeration is ill-formed. It is implementation-defined which integral type is used as the underlying type ~~for an enumeration~~ except that 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.~~

7 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~~F~~or 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}.82) It is possible to define an enumeration that has values not defined by any of its *enumerator*s.

8 Two enumeration types are layout-compatible if they have the same *underlying type*.

9 The value of an *enumerator* or an object of a~~n~~ 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*]

10 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.

11 ~~The~~Each enum-name and each POE enumerator~~ declared by an enum-specifier~~ is declared in the scope that immediately contains the enum-specifier. Each non-POE enumerator is declared in the scope of the enumeration. 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
}

enum class altitude { high='h', low='l' };

void h()
{
        altitude a;                 // OK
        a = high;                   // error: high not in scope
        a = altitude::low;          // 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

Add the following new subclause (modeled on 3.3.6 [basic.scope.class]):

3.3.6a Enumeration scope **[basic.scope.enum]**

1 The following rules describe the scope of names declared in non-POE enumerations.

1) The potential scope of a name declared in a non-POE enumeration consists of the declarative region following the name's point of declaration.

2) A name N used in a non-POE enumeration E shall refer to the same declaration in its context and when re-evaluated in the completed scope of E. No diagnostic is required for a violation of this rule.

3) The name of a non-POE enumerator shall only be used in the scope of its enumeration, or after the :: scope resolution operator (5.1) applied to the name of its enumeration.

2 The name of a non-POE enumerator shall only be used as follows:

— in the scope of its enumeration,

— after the :: scope resolution operator (5.1) applied to the name of its enumeration.

In §4.5(2), §4.7(1), §4.9(2), §4.12(1), change "enumeration" (or "enumerated") to "POE enumeration".

In §7.1.5.3, change one production of *elaborated-type-specifier* as follows:

elaborated-type-specifier:

…
~~enum~~<u>*enum-key*</u> $::_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
…

In §7.1.5.3(3) (two places), change "enum keyword" to "*enum-key*".

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).

[Sutter04] H. Sutter and D. Miller. "Strongly Typed Enums" (ISO/IEC JTC1/SC22/WG21 N1579 = ANSI/INCITS J16 04-0019).