Improving Enumeration Types [N1513=03-0096]

David E. Miller

Introduction

It has been said, "C enumerations constitute a curiously half-baked concept." [Design and Evolution of C++ (C) 1994, B.Stroustrup, p.253] Even though C++ enumerations have some improvements over C, some issues remain. This submission endeavors to improve enumerations by eliminating certain lingering deficiencies.

Summary of issues

1. Inconsistent relationship between enumerations of separate types
2. Limitation to integral types
3. Inability to specify actual storage space and type to be used
4. Potentially conflicting names, unless separated by namespace use
5. Denormalized tables mapping enumeration values to printable names and vice versa

Issues and proposed changes

1. Inconsistent relationship between enumerations of separate types

Statement of problem

Although it is not permitted to assign from one enumeration type to another, it is permitted to compare such values, in that the values are silently converted to integral types.

Example of problem

```
typedef enum Color { ClrRed, ClrOrange, ClrYellow,
ClrGreen, ClrBlue, ClrViolet };

typedef enum Alert { CndGreen, CndYellow, CndRed };

Color c1 = ClrRed ;

Alert a1 = CndGreen ;

a1 = c1 ; // not allowed

a1 = ClrYellow ; // not allowed

bool goToAlertAndArmWeapons = ( a1 >= ClrYellow ) ; //
Allowed, but silently incorrect
```

Current work-around

```
class PseudoEnumColor // class simplified for clarity

{

   typedef enum Color { Red, Orange, Yellow, Green, Blue,
Violet }; // private

public:

   static const PseudoEnumColor ClrRed, ClrOrange,
ClrYellow, ClrGreen, ClrBlue;

   Color m_value ;

   explicit PseudoEnumColor( Color value ): m_value( value
){}

   bool operator<( PseudoEnumColor const & other ) { return
this->m_value < other.m_value ; }

   int getIntValue() const { return m_value ; }

};

const PseudoEnumColor PseudoEnumColor::ClrRed(
PseudoEnumColor::Red );

// Similarly for Alert

PseudoEnumAlert a1 = PseudoEnumAlert::ClrGreen ;

bool goToAlertAndArmWeapons = ( a1 >= Clr***Yellow ) ; // No
longer allowed, with or without scoping ClrYellow
```

Proposed solution

Extending the "explicit" keyword to enumeration types can eliminate the implicit conversion without breaking any existing code.

Example of proposed solution

```
typedef explicit enum Color { ClrRed, ClrOrange, ClrYellow,
ClrGreen, ClrBlue, ClrViolet };

typedef explicit enum Alert { CndGreen, CndYellow, CndRed
};

Alert a1 = CndGreen ;

Color c1 = ClrRed ;
```

```
a1 = c1 ; // not allowed

a1 = Clr Yellow ; // not allowed

bool goToAlertAndArmWeapons = ( a1 >= Clr Yellow ) ; // No
longer allowed, due to specification of "explicit"
```

Effect on existing code

Since the change involves adding the "explicit" keyword to get the desired effect, no prior code is affected.

Ease of explanation

Since the change is similar in tone to the extant use of "explicit," differing mainly in direction, i.e. "explicit" for constructors blocks implicit conversion *to* the type, whereas the proposed use blocks implicit conversion *from* the type, it should be very easy to explain and to use.

2. *Limitation to integral types*

Statement of Problem

For a certain purposes, it would be useful to allow floating point enumerations.

Although it is common to consider enumerations to be specific values, as listed in the declarations, they are also used to represent bit combinations and ranges, among other purposes. It is convenient to be able to specify floating point values as enumerations, particularly as values to be passed to functions, which by being declared to take floating enumeration parameters, can be protected at compilation time against invalid values.

Example of problem

```
double const valid1 = 1.234, valid2 = 2.468;

void func( double val ); // val must be limited to those
values specified outside the function

func( valid1 ); // value is allowed

func( 3.333 ); // value should not be allowed
```

Current work-arounds

Various work-around exist involving use of class wrappers to preclude casually passing values not prespecified. As they are fairly obvious, they are not presented here.

Proposed solution

Extending the allowed constants for enum types can extend enumeration types without breaking any existing code.

The underlying data type of the enumeration would be the least required to hold all named values, i.e. if no double value were specified, the underlying data type would be float.

Example of proposed solution

```
typedef enum FloatEnum { LowVal = 1.23, MidVal = 4.56,
HighVal = 7.89 };

void func( FloatEnum val );

func( LowVal ); // allowed

func( 3.333 ); // ordinary float value would be blocked at
compilation time
```

Effect on existing code

Since the change a superset of currently allowed values, no prior code is affected.

Ease of explanation

Since the change is a simple extension of allowed types to allow floating point, it should be very easy to explain and to use.

3. Inability to specify actual storage space and type to be used

Statement of Problem

There are actually two related problems: the need to be able to know, definitely, how much space will be used by an enumeration variable, particularly in a packed struct, and the need to be able to specify how that enumeration will be treated when used as a number, e.g. as signed or unsigned.

There are times when it is essential 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. Because the specification of enumeration types allows implementations to take either the minimal space necessary or a larger amount, they cannot be used reliably in such structs.

Examples of problem

Size consistency and minimization

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

struct Packet

{

   Version m_version ; // bad, size can vary by
implementation
```

```
   Version getVersion() const { return m_version ; }
   void    setVersion( Version ver ) { m_version = ver ; }
};
```

Type specification

```
#include <iostream>
using namespace std ;
typedef enum E1_t { E1a = 1, E1b = 2, Ebig = 0xFFFFFFF0U };
int main()
{
   E1_t e1 = E1a ;
   cout << "sizeof( e1 ) = " << sizeof( e1 )
        << ", sizeof( E1_t ) = " << sizeof( E1_t )
        << endl ;
   cout << "E1a = " << E1a << ", Ebig = " << Ebig << endl ;
   if( E1a >= -1 )
      cout << "E1a >= -1" ;
   else
      cout << "E1a < -1" ;
   cout << endl ;
   if( Ebig >= -1 )
      cout << "Ebig >= -1" ;
   else
      cout << "Ebig < -1" ;
   cout << endl ;
   return 0 ;
}
// program output
```

```
sizeof( e1 ) = 4, sizeof( E1_t ) = 4

E1a = 1, Ebig = -16

E1a >= -1

Ebig < -1

A = 111
```

This result (treating all E1_t values, especially Ebig, as signed) is counter-intuitive, particularly to the naive user, who declared Ebig using a constant ending in a suffix specifying unsignedness and expected the compiler to understand the intent.

Current work-arounds

Size specification work-around

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

struct Packet

{

   unsigned char m_version ; // works, but requires casting

   Version getVersion() const { return (Version)m_version ;
}

   void    setVersion( Version ver ) { m_version =
(unsigned char)ver ; } // Not strictly needed, but helps
turn off compiler warnings about type change

};
```

Additionally, internal class uses of the unsigned char member may require either casts or access functions.

Type workarounds

Class wrappers

Explicit casts

Proposed solution

Borrowing from a C# approach, it is suggested there be an option to specify the underlying data type, which would also be the type to which implicit numeric conversions would be made. When utilized, this option would supersede the current size and type specifications.

Specifying a type inadequate to hold all the listed values would not be permitted.

Example of proposed solution

```
typedef enum E1_t: unsigned int { E1a = 1, E1b = 2, Ebig =
0xFFFFFFF0U };

// typedef enum Version: signed char { Ver1 = 1, Ver2 = 2
};

typedef enum { Ver1 = 1, Ver2 = 2 } VersionSChar: signed
char, VersionInt: int ; // Note this would allow related
enums of different sizes, useful for some optimizations

struct Packet

{

   VersionSChar m_version ; // size (and alignment)
guaranteed to be identical to that of signed char

   VersionSChar getVersion() const { return m_version ; }
// No cast needed

   void setVersion( VersionSChar ver ) { m_version = ver ;
}

};
```

Effect on existing code

Since the new capability is optional, does not change the semantics of any existing code, and requires the user to add code not currently valid, no prior code should be affected.

Ease of explanation

Since the change is a simple extension of allowed types to allow floating point, it should be very easy to explain and to use.

Because the new feature makes the underlying data type and numeric conversion explicit, it should be easier to teach than the current rules.

4. *Potentially conflicting names, unless separated by namespace use; potentially confused even with namespace use.*

Statement of problem

In a single scope, declaring two enumeration value names results in a conflict.

Declaring two enumerations carrying identical value names in different namespaces, but inadvertently specifying the wrong "using," can result in the wrong enumeration constant being used.

Example of problem

```
namespace NS1 {

typedef enum Color { Red, Orange, Yellow, Green, Blue,
Violet };

};

namespace NS2 {

typedef enum Alert { Green, Yellow, Red };

};

using namespace NS1 ; // but NS2 was what should have been
specified, if anything

// Alternatively, another enum with conflicting names could
be declared within a scope

Alert a1 = NS2::Green ;

Color c1 = Red ;

a1 = c1 ; // not allowed

a1 = Yellow ; // not allowed

bool goToAlertAndArmWeapons = ( a1 >= Yellow ) ; //
```
Allowed, but likely incorrect

Current work-around

```
class PseudoEnumColor // class simplified for clarity

{

   typedef enum Color { Red, Orange, Yellow, Green, Blue,
Violet }; // private

public:

   static const PseudoEnumColor ClrRed, ClrOrange,
ClrYellow, ClrGreen, ClrBlue;

Color m_value ;

   explicit PseudoEnumColor( Color value ): m_value( value
){}

   bool operator<( PseudoEnumColor const & other ) { return
this->m_value < other.m_value ; }

   int getIntValue() const { return m_value ; }
```

```
};

const PseudoEnumColor PseudoEnumColor::ClrRed(
PseudoEnumColor::Red );

// Similarly for Alert

PseudoEnumAlert a1 = PseudoEnumAlert::ClrGreen ;

bool goToAlertAndArmWeapons = ( a1 >= ClrYellow ) ; // No
longer allowed, with or without scoping of ClrYellow
```

Proposed solution

Allowing the typedef name to be used as a scope modifier to the enumeration constants, similarly to an approach used in C#.

Example of proposed solution

```
typedef [explicit] enum Color { Red, Orange, Yellow, Green,
Blue, Violet };

typedef [explicit] enum Alert { Green, Yellow, Red };

// Note it would be possible to have non-unique enumeration
constants in the same scope.

Alert a1 = Alert.Green ; // or Alert::Green ;

Color c1 = Color.Red ; // or Color::Red

a1 = c1 ; // not allowed

a1 = Yellow ; // not allowed, due to ambiguity

bool goToAlertAndArmWeapons = ( a1 >= Yellow ) ; // No
longer allowed, due to ambiguity (leaving aside the
possible use of proposed "explicit")
```

Effect on existing code

Since the change involves adding to the enumeration value name a prefix not currently allowed, there should be no effect on existing valid code. However, if the "::" version is used, there may be a possibility of changing code in the event the same name is used for both a namespace and an enumeration type, though this is probably not found in actual practice.

Ease of explanation

Since the proposed prefix acts as a disambiguator, its usage should be clear, even to a novice.

There is an additional benefit in making it easier for printable names for enumerations to more closely match the internal names, since there would be less need for disambiguating prefixes.

5. *Denormalized tables mapping enumeration values to printable names and vice versa*

Statement of problem

It is common to need functions to convert back and forth between enumeration values and printable strings corresponding to the names. Although it is not hard to write such functions, the mappings amount to a denormalized database, with the obvious maintenance issues associated with insuring the tables match exactly whenever a new enumeration value is added or removed.

Additionally, this common functionality is implemented in various ways lacking any standardization as to name or exact functionality.

Example of problem

```
typedef enum Alert { Green, Yellow, Red };

Alert valueFromName( char const pszName )

{

   Alert valueCorrespondingToName ;

   // compare pszName to all possible names

   if( noMatch )

      throw .... ;

   return valueCorrespondingToName ;

};

char const * nameFromValue( Alert val )

{

   switch( val )

   {

      case CndGreen: return "Green" ;

      ...

      default: return "?" ; // or throw something, e.g. val

   }
```

```
}

char const * szAlertGreen[] = "Green" ;

Alert a1FromName = valueFromName( szAlertGreen );

char const * pszAlertFromVal = nameFromValue( a1FromName );
```

Current work-around

Code is currently written in an ad hoc manner, with maintenance often done manually, rather than in an automatically guaranteed fashion, except in those places utilizing code generators.

Proposed solution

Borrowing from C#, the language should specify the conversion functions' names and algorithms, including what to do when input is invalid. To avoid name collisions, it is suggested that the enumeration type be used as a disambiguating scope, similarly to C# usage is this context.

In the event more than one enumeration constant within a particular enumeration carried the same value, translation from value to name would convert to the first lexically defined name.

Example of proposed solution

```
typedef [explicit] enum Alert { Green, Yellow, Red };

char const * szAlertGreen[] = "Green" ;

Alert a1FromName ;

bool success = Alert::valueFromName( szAlertGreen, &a1FromName
); // No longer necessary to declare the function

char const * pszAlertFromVal = Alert::nameFromValue( a1FromName
); ); // No longer necessary to declare the function
```

In the event functions of the same name and parameter type were defined, those definitions would pre-empt the default implementations, analogously to the specification of copy constructors and assignment operators in classes.

If not used, the functions need not be generated.

Effect on existing code

Since the proposed function names have scope prefixes that are currently invalid syntax, there should be no effect upon existing code.

Ease of explanation

The extension is very similar to C# enumeration functionality and appears very simple and straightforward to explain to novices, particularly inasmuch as it would allow them to write

cleaner code than is often the case, at the stage before novices are consider look-up and error checking to be second nature.

Conclusion

The changes described are intended to reduce the likelihood of undetected error while enabling code to be written more clearly and consistently. The proposed changes would do so without breaking existing code.