C++ Scoped Enum Enhancements

ALEXANDER CHRISTENSEN

Contents

1 INTRODUCTION

This proposal is the first revision of "C++ Scoped Enum Proposal", which was circulated in the C++ proposals email list during the winter of 2021/2022. The goal of writing this proposal is to gain feedback and insights, which may then be used for further revisions.
Note: This proposal targets only scoped enums, and does not provide any effort to address *unscoped* enums.

Given my limited experience (actually: none) in writing C++ proposals, I know what I would like the proposal to address and introduce to the core language, but not necessarily how it should be done. The suggested notations provided are, thus, merely suggestions - and not concrete proposals for a concious choice of syntax.

Author's address: Alexander Christensen, <alex_c007@hotmail.com>.

2 MOTIVATION AND SCOPE

The initial motivation for this proposal was the lack of a good way in the standard library to convert an enum value to a proper string representation. Very often, I have found a need to log an enum for various purposes, and every time I create a new enum type I have to write such a function again. An example:

```
1  enum class GraphicsApiType { none, opengl, vulkan };
2
3  const char* get_api_type_string(GraphicsApiType apiType) {
4      switch (apiType) {
5      case GraphicsApiType::none:   return "none";
6      case GraphicsApiType::opengl: return "opengl";
7      case GraphicsApiType::vulkan: return "vulkan";
8      default: return "<unrecognized>";
9      }
10 }
```

This is cumbersome to maintain, for every time a value is added or modified inside the enum type, this other function has to be modified as well. But besides this, a potential *run-time* error may be introduced when an invalid enum is provided.

A session of browsing around sites such as StackOverflow revealed sometimes quite vividly imaginary answers for how to circumvent this limitation in the language:

(1) https://stackoverflow.com/questions/11421432/how-can-i-output-the-value-of-an-enum-class-in-c11
(2) https://stackoverflow.com/questions/1390703/enumerate-over-an-enum-in-c
(3) https://stackoverflow.com/questions/6281461/enum-to-string-c
(4) https://stackoverflow.com/questions/201593/is-there-a-simple-way-to-convert-c-enum-to-string

Quite intuitively, two key observations were made:

- This proposal introduces changes to the core language.
- While fixing enum to string functionality, there's a lot more that we could do at the same time.

2.1 Scope for this proposal

Only *scoped enums* are extended with this proposal, because they are fully type safe, meaning that we can theoretically perform all necessary computations needed for this proposal *compile-time*, with no risk of throwing exceptions or using any other runtime mechanism.

This, at least was the aim. Added is one particular *exception* to this, where we try to convert to a scoped enum from its underlying type. This will be described in detail.

3 IMPACT ON THE STANDARD

This proposal requires some features that as of this writing cannot be covered by writing a library. These features *might* possibly be implemented by the compilers instead of modifying or extending the C++ standard. This is preferred.

4 PROPOSED FEATURES

4.1 Concept for "is-an-enum"

The basis for this proposal is a concept for restricting correct type substitution on templated types. This can be implemented fully as a *library feature*, using C++23:

```
namespace std {
    template<typename T>
    concept scoped_enum = std::is_scoped_enum_v<T>;
}

template<std::scoped_enum T>
class foo { int bar; };

// Try it out:
foo<GraphicsApiType> f1;  // <-- success!
foo<unsigned long> f2;    // <-- fails to compile!
```

4.2 Convert scoped enum to string

Two ways of obtaining a string is proposed: A simple one which contains only the name of the enum value, and a fully-qualified which *additionally* contains namespaces, classes, and other scopes:

```
namespace std {
    template<std::scoped_enum T>
    constexpr std::string_view enum_str(T t) { /* compiler implements this */ }
}

std::cout << std::enum_str<GraphicsApiType::opengl>() << std::endl;
// --> "opengl"
std::cout << std::enum_str_full<GraphicsApiType::opengl>() << std::endl;
// "graphics::GraphicsApiType::opengl"
```

The return type itself gives us a number of possibilites, but two are probably worthy of consideration: `const char*` and `std::string_view`. It is probably also worth considering what notation is *optimal*. In the example above, we have a templated function which return a `constexpr` string representation.

Note: I am unsure how best to write this.

4.3 Obtain the underlying type

In current C++, when needing to compare an enum value to its underlying type, a relatively verbose expression is needed which involves a "`static_cast`" and possible also an "`std::underlying_type`". This has been somewhat mitigated by the introduction of `std::to_underlying`, available with C++23, but this proposal adds a new template meta-function which does almost the same thing:

```
template<scoped_enum E>
struct enum_type {
```

```
    using type = std::underlying_type<E>::type;
};

template<scoped_enum E>
using enum_type_t = std::enum_type<E>::type;

// Try it out:
enum class Color : int { red, green, blue, yellow };
std::enum_type_t<Color> yellow = Color::yellow;
```

The benefit of this meta-function, as opposed to `std::to_underlying`, is that this meta-function does not work for unscoped enumerations. This is because `std::to_underlying` uses `std::underlying_type`, while this new meta-function uses the newly proposed concept `std::scoped_enum`. This creates a more accentuated difference between scoped and unscoped enums, and thereby encourages the use of scoped enums for enhanced type safety and scope safety.

4.4 Get the underlying value

Similar to how getting an underlying type can be tedious, this proposal also addresses the need for obtaining the underlying value of a scoped enum value:

```
template<scoped_enum Enum>
constexpr std::enum_type_t<Enum> enum_value(Enum e) noexcept
{
    // TODO: Error - use enum_cast instead!
    return static_cast<std::enum_type_t<Enum>>(e);
}
```

4.5 Safe enum conversion (language feature)

Sometimes, a conversion from an underlying type to an enum value may be approved without being valid. And there exists no compile-time feature to check for this. An example of how it may go wrong:

```
enum class MyScopedEnum : int
{
    value_0 = 0, value_1 = 1, value_2 = 2, value_3 = 3, value_43 = 43, value_57 = 57
};

constexpr MyScopedEnum test = static_cast<MyScopedEnum>(56);
```

Even if we explicitly denote our variable `constexpr`, the compiler is not able to enforce correct behaviour, and so we obtain undefined behaviour. The topic is discussed on StackOverflow:
https://stackoverflow.com/questions/17811179/safe-way-to-convert-an-integer-in-an-enum

This proposal thus defines a template meta-function which fails at compile-time if the conversion is invalid. It might be possible to also define a *run-time* mechanism which throws an exception, but this revision does not discuss that topic further (for now). Proposed syntax:

```
template<std::scoped_enum T>
constexpr bool is_enum_convertible(std::enum_type_t<T> t) {
    return ...; // <-- compiler implements this
};

template<std::scoped_enum T>
consteval T enum_cast(std::enum_type_t<T> t) {
    static_assert(std::is_enum_convertible<T>(t));
    return static_cast<T>(t);
}
```

4.6 IMPROVED Safe enum conversion (language feature)

Current C++, `static_cast` is unsafe.

```
enum class MyScopedEnum : int
{
    val_0 = 0, val_1 = 1, val_2 = 2
};

// FAIL: Compiler thinks this is okay
MyScopedEnum e = static_cast<MyScopedEnum>(4);
```

Proposed restriction to `static_cast`.

```
enum class MyScopedEnum : int
{
    val_0 = 0, val_1 = 1, val_2 = 2
};

// GOOD: Compiler throws an error
MyScopedEnum e = static_cast<MyScopedEnum>(4);
```

4.7 Iterating through a scoped enum

Another interesting scenario is if an enum contains "identifiers", where for each we want to create some data structure and add that to some general storage. An example of such storage is shown below, using the `scoped_enum` concept proposed earlier:

```
template<std::scoped_enum E, typename T>
class EnumMap {
public:
    bool has_value(E e) { return mEnumMap.contains(e); }
    void add_value(E e, T&& t) { mEnumMap[e] = t; }
    std::optional<T> get_value(E e) {
        if (auto it = mEnumMap.find(e); it != mEnumMap.end()) {
            return { it->second };
        } else { return std::nullopt; }
    }
private:
    std::map<E, T> mEnumMap;
};
```

We can then obtain a "bi-directional" iterator, which allows us to iterate through all values in the enumeration and add them to the storage:

```
void fill(EnumMap<auto T, auto V>& em) {
    for (T t : std::enum_values<T>)
        em.add value(t, V{});
}
```

4.8 Enum Cardinality

Another feature which might sometimes be useful, is the ability to count the number of values inside an enumeration. A real-world example, here found in the `glslang` library for shader compilation:

```
typedef enum {
    EShSourceNone,
    EShSourceGlsl,               // GLSL, includes ESSL (OpenGL ES GLSL)
    EShSourceHlsl,               // HLSL
    LAST_ELEMENT_MARKER(EShSourceCount),
} EShSource;                     // if EShLanguage were EShStage, this could be EShLanguage instead
```

Granted, this is a `C` library, but the use case for `C++` would be identical. And it would likely be used for validating the input because `enums` are unsafe:

```
void my_function(EShSource e) {
    if (e < 0 || e >= LAST_ELEMENT_MARKER) {
        // report an error...
    }
}
```

This proposal then adds a new template meta-function, which at compile time computes the number of elements - ie. the *cardinality* - of a scoped enumueration:

```
template<scoped_enum E>
struct enum_cardinality {
    static constexpr std::size_t value = 1; // <-- compiler implements this
};

template<scoped_enum E>
std::size_t enum_cardinality_v = std::enum_cardinality<E>::value;

// Try it out:
enum class Weekday { monday, tuesday, /* etc. */ };
static_assert(std::enum_cardinality_v<Weekday> == 7);
```

The example from before could be written, then, in a manner both type safe and value safe, with no scope leakage:

```
enum class EShSource : int {
    EShSourceNone = 0, EShSourceGlsl, EShSourceHlsl
};
```

```
4
5  void my_function(EShSource e) {
6      if (std::enum_value(e) < 0 || std::enum_value(e) >= std::enum_cardinality_v<EShSource>) {
7          // report an error...
8      }
9  }
```

And this leads us to the next topic..

4.9 Enum bounds checking

It's annoying to write the statement in `my_function` above, so there should also be a shorthand notation for that, thusly proposed:

```
1
2  enum class EShSource : int {
3      EShSourceNone = 0, EShSourceGlsl, EShSourceHlsl
4  };
5
6  void my_function(EShSource e) {
7      if (std::enum_value(e) < 0 || std::enum_value(e) >= std::enum_cardinality_v<EShSource>) {
8          // report an error...
9      }
10 }
```

NOTE: We absolutely MUST test that enums are not given custom values, ie. only default in range [0, cardinality-1], OTHERWISE bounds checking cannot work properly!

4.10 Enum position

We might also want to know the position or the *index* of an enum value inside an enumeration. As with `std::enum_cardinality`, we need to rely on the compiler to provide an implementation, as well as a requirement of strict compile-time evaluation:

```
1  template<std::scoped_enum T, T E>
2  struct enum_position {
3      static constexpr std::size_t value = 0; // <-- compiler implements this
4  };
5
6  template<std::scoped_enum T, T E>
7  std::size_t enum_position_v = std::enum_position<T, E>::value;
8
9  // Try it out:
10 enum class Fruit { banana = 1, apple = 2, orange = 4, clementine = 8 };
11 static assert(std::enum_value_v<Fruit::orange> == 4);
12 static assert(std::enum_position<Fruit, Fruit::orange> == 2);
```

With both `enum_cardinality` and `enum_position` implemented, we can create e.g. create a priority mapper with a priority for each enum value:

```cpp
template<std::scoped_enum T>
class EnumPriorityMapper {
public:
    unsigned int GetPriority(T t) {
        return mPriorities[std::enum_position_v<T, t>];
    }
    void SetPriority(T t, unsigned int p) {
        mPriorities[std::enum_position_v<T, t>] = p;
    }
private:
    unsigned int mPriorities[std::enum_cardinality_v<T>];
};
```

4.11 Weak ordering

With a good comparison operator in place, it becomes straight-forward to test the order in which values appear inside an enumeration. Using the "spaceship operator" introduced with C++20, this proposal defines two three-way comparison functions:

```cpp
template<std::scoped_enum T>
constexpr std::weak_ordering operator<=>(T t, std::integral auto i) {
    return std::enum_value(t) <=> i;
}

template<std::scoped_enum T>
std::weak_ordering operator<=>(T s, T t) {
    return std::enum_value(s) <=> std::enum_value(t);
}

// Try it out:
enum class MyScopedEnum : int {
    value_0 = 0, value_1 = 1, value_2 = 2, value_3 = 3, value_43 = 43, value_57 = 57
};
static_assert((MyScopedEnum::value_0 <=> 0) == 0);
static_assert((1 <=> MyScopedEnum::value_2) < 0);
static_assert((MyScopedEnum::value_2 <=> 1) > 0);
```

4.12 Enum as bitmask

Bitmasking enums is a larger topic, one that has always been kind of awkward. We can easily use C-style enums and directly mask them because they are integers, but this approach provides no safety guarantees. The following example shows such an awkward example:

```cpp
enum MyUnscopedEnum { val_0 = 0, val_1 = 1, val_4 = 4 };
enum AnotherUnscopedEnum { val_5 = 5, val_6 = 6 };
```

```
3
4  // C++20 deprecates this, but for earlier versions (<= 17) it is valid.
5  int m = val_0 | val_5;
```

At least, this is the output from GCC (I have not tested with other compilers!). But we can improve upon the type safety a bit by using a scoped enum instead. But now we have another awkward problem - we must statically cast to underlying type before using the bitwise operator:

```
1  enum class MyScopedEnum { val_0 = 0, val_1 = 1, val_4 = 4 };
2  enum class AnotherScopedEnum { val_5 = 5, val_6 = 6 };
3
4  int p = MyScopedEnum::val_0 | AnotherScopedEnum::val_5; // error: no match for 'operator|'
5  int q = static_cast<int>(MyScopedEnum::val_0) | static_cast<int>(AnotherScopedEnum::val_5); // valid
6  int q = std::enum_value(MyScopedEnum::val_0) | std::enum_value(AnotherScopedEnum::val_5); // same
```

We could define an `operator|` as suggested by the compiler, but then we are left with another very awkward problem:

```
1  template<std::scoped_enum T>
2  constexpr ?? operator| (T t1, T t2) {
3      std::enum_type_t<T> result = std::enum_value<T>(t1) | std::enum_value<T>(t2);
4      return ??
5  }
```

Now we don't know which type to return - there are two options, and both of them are bad. If we return the underlying (integer) type then we bypass type safety, and if we return the same enum type then we return a value that should not exist:

```
1  enum class MyFlags : int { val_1 = 0x01, val_2 = 0x02, val_4 = 0x04, val_8 = 0x08 };
2
3  MyFlags f1 = MyFlags::val_1 | MyFlags::val_2; // This is not a valid 'MyFlags' !
4  int f2 = MyFlags::val_1 | MyFlags::val_2 | MyFlags::val_4; // Problematic!
```

The second case in particular is problematic, because it bypasses type safety. When I tested this I found that my compiler evaluates from right to left, that is it will evaluate "`val_1 | (val_2 | val_4)`". But whatever the case, if the operator returns an integer, then we must define another operator which takes as arguments a scoped enum and an integer, and that enables us to pass to it any integer, like -1!
Both choices yield bad code, or at the very least *unsafe* code, and it would be far better if the process of bitmasking enums could be automated in a way that is type safe, practical, without overhead, and adheres to the general design philosophies of the standard library. It would also be very nice if the operators supported `constexpr`. The first solution that comes to mind is to create a class. The solution might not be perfect, and likely needs some adjusting, but it does solve the problems described above while being relatively easy to use:

```
1  template<typename T>
2  requires enum_bitmask_type<T>
```

```
class enum_bitmask {
    using type = std::enum_type_t<T>;

public:
    bool empty();
    void add_flag(T _t);
    bool has_flag(T _t);
    void clear();

    type operator()();
    bool operator& (T _t);
    enum_bitmask operator| (T _t);
    enum_bitmask& operator|= (T _t);
    bool operator== (enum_bitmask& b);

private:
    type t {0U};
};
```

This class may be implemented in roughly 50 lines of code, plus some additional operators for parameter switching. Below is a short code snippet (which has been tested) that showcases some example usage:

```
enum class MyBitflag : unsigned int {
    val_1 = 0x01U, val_2 = 0x02U, val_4 = 0x04U, val_8 = 0x08U
};

constexpr std::enum_bitmask<MyBitflag> test1;
static_assert(test1.empty());

constexpr std::enum_bitmask<MyBitflag> test2 =
    MyBitflag::val_1 | MyBitflag::val_2 | MyBitflag::val_4 | MyBitflag::val_8;
static_assert(test2.has_flag(MyBitflag::val_4));
```

The examples are hopefully enough to express the *intent* of the class. The full implementation is included in the appendix(REF??!). The way the class and its operators are designed allow for compile-time checks through `static_cast`, and lets us use scoped enums as fully type-safe bitmasking operands.

The four public member functions provide an interface that is easy to read and use, and by using a template we omit the penalties of vtable lookups. The templated type which must be a scoped enum is verified by a "concept", because additional constraints should be placed upon usage of the class. One such constraint would be to check that the scoped enum used does not have a value that is zero:

```
template<scoped_enum T>
struct enum_contains_zero {
    static constexpr bool value; // Should be implemented in the compiler!
};
```

```
5
6  template<scoped_enum E>
7  using enum_contains_zero_v = std::enum_contains_zero<E>::value;
```

Another constraint might be to require that the underlying type of the enum is unsigned, unless that would yield some platform-specific problems (I'm not sure!).

5 PROPOSED NOTATION SUMMARY

In summary, there's a lot of new features that would *significantly* improve the usage of scoped enums. Some of these may be implemented as library features, while others simply cannot be written with current C++. Since most of the symbols defined are dependent upon `std::is_scoped_enum`, C++23 is the *minimal* requirement for an implementation of this proposal. A quick overview of the function and meta-function signatures are listed here:

```
1  #include <type_traits>
2
3  namespace std {
4      template<typename T>
5      concept scoped_enum = std::is_scoped_enum_v<T>;
6
7      template<scoped_enum E>
8      struct enum_cardinality {
9          static constexpr std::size_t value; // Implementation-defined
10     };
11
12     template<scoped_enum E>
13     std::size_t enum_cardinality_v = std::enum_cardinality<E>::value;
14
15     template<std::scoped_enum T, T E>
16     struct enum_position {
17         static constexpr std::size_t value = 0; // Implementation-defined
18     };
19
20     template<std::scoped_enum T, T E>
21     std::size_t enum_position_v = std::enum_position<T, E>::value;
22
23     template<scoped_enum E>
24     struct enum_type {
25         using type = std::underlying_type<E>::type;
26     };
27
28     template<scoped_enum E>
29     using enum_type_t = std::enum_type<E>::type;
30
31     template<scoped_enum Enum>
32     constexpr std::enum_type_t<Enum> enum_value(Enum e) noexcept {
```

```
33          return static_cast<std::enum_type_t<Enum>>(e);
34      }
35  } // namespace std
36
37  template<std::scoped_enum T>
38  constexpr std::weak_ordering operator<=>(T t, std::integral auto i)
39  {
40      return std::enum_value(t) <=> i;
41  }
42
43  template<std::scoped_enum T>
44  std::weak_ordering operator<=>(T s, T t)
45  {
46      return std::enum_value(s) <=> std::enum_value(t);
47  }
48
49  template<std::scoped_enum T>
50  constexpr std::enum_type_t<T> operator| (T s, T t)
51  {
52      return std::enum_type_t<T>(s) | std::enum_type_t<T>(t);
53  }
54
55  template<std::scoped_enum T>
56  constexpr bool operator& (std::enum_type_t<T> f, T t)
57  {
58      return f & std::enum_value(t);
59  }
60
61  template<std::scoped_enum T>
62  constexpr bool operator& (T t, std::enum_type_t<T> f)
63  {
64      return std::enum_value(t) & f;
65  }
```

6 DESIGN DECISIONS

The overall design goals have been twofold:

(1) Provide enhancements that all operate compile-time.
(2) Only provide enhancements for the type-safe scoped enums. Do nothing for the "old" C-style enums.

As such, most features have been written with the proposed syntax of template meta-functions, with the use of modern C++ features such as `concepts` and "three-way comparison" (<=>). The proposed syntax are merely *suggestions*, ie. "what I want", and not necessarily "how I want it".