# C++ Scoped Enum proposal

Alexander Christensen

November 24, 2021

Following discussions on StackOverflow, and the interest generated, it seems that there could be motivation for extending the scoped enums to allow for more flexible operations. Such questions, and their often vividly imaginary answers, are found here:

1. how-can-i-output-the-value-of-an-enum-class-in-c11
2. enumerate-over-an-enum-in-c
3. enum-to-string-c
4. is-there-a-simple-way-to-convert-c-enum-to-string

Following the apparent interest in the subject, this proposal aims to bring into the C++ (>= 20) standard library improved *compile-time* support for scoped enums (as signified by "`enum class`"), in particular enumeration, and cast to string or underlying type. As such, `type_traits` or other RTTI-techniques will not be considered.

For maintaining backwards-compatibility with C, this proposal *only* targets the scoped enums, and **not** the C-style (unscoped) enums.

## 1 Enumeration

Sometimes it may be desirable to enumerate through a scoped enum to know its values. One particular use case is when an enum is used as a template parameter:

```
template<"enum" T, typename V>
requires std::is_default_constructible_v<V>
class foo {
public:
  bool has_value(T t) { return mEnumMap.contains(t); }
  void add_value(T t, V&& v) { mEnumMap[t] = v; }
private:
  std::map<T, V> mEnumMap;
};

void fill(foo<auto T, auto V>& f) {
  auto begin = std::enum_iterator<T>.cbegin();
  auto end = std::enum_iterator<T>.cend();
  for (auto it = begin; it != end; it++)  f.add_value(*it, V{});

  // Alternatively:
  for (auto t : std::enum_values_v<T>)  f.add_value(t, V{});
}
```

In such a system, we may maintain a dictionary of various enum members and bind them to a particular type `"V"`.

# 2 Underlying type

In current C++ standard, when needing to compare an enum value with its underlying type, a relatively verbose code expression is required:

```
// Check if enum value is 0
auto my_enum_value = MyEnum::Value;
bool is_zero = static_cast<std::underlying_type_t<MyEnum>>(my_enum_value) == 0;
```

This document proposes a simpler syntax. Assume we have a definition "`enum class MyEnum`":

```
// Check if enum value is 0
auto my_enum_value = MyEnum::Value;
bool is_zero = std::enum_value<my_enum_value> == 0;
```

Seemingly, this does not yield much benefit since we just removed a `static_cast` and replaced one template meta-function with another. But without the `static_cast` we can have the compiler auto-generate an implicit operator overload, similar to "`ToInt()`" found in some object-oriented languages. And with that in mind, we should be able to write our expression above in much simpler terms:

```
// Check if enum value is 0
auto my_enum_value = MyEnum::Value;
bool is_zero = (my_enum_value == 0);
```

Sometimes, we may also wish to convert the other way around. But for this we need to make sure that our input value does actually represent a valid enum value. Additionally, this document proposes the following syntax:

```
enum class Color : int { red = 0, green, blue };

optional<Color> newColor = enum_cast<Color>(3);
static_assert(newColor == std::nullopt);
```

# 3 Scoped enum cardinality

In some usage scenarios it can be useful to retrieve the number of values declared inside a scoped enum. Current possibilities includes only hacks, so the following syntax is proposed:

```
enum class Weekdays { monday, tuesday /*, etc. */ };

// NOTE: Possible alternative namings - "enum_count_v" or "enum_size_v".
static_assert(std::enum_cardinality_v<Weekdays> == 7);
```

# 4 String representation

Sometimes, for debugging purposes, we might want to be able to output to a terminal the string name of an enum value. Instead of creating such a function ourselves and maintaining it when new values are added to the enum or renamed, better yet to let the compiler do the heavy-lifting for us. Proposed syntax:

```
enum class Color { red, green, blue };

// Two alternatives:
std::cout << std::enum_string<Color::red>() << std::endl;
std::cout << std::enum_string_v<Color::red> << std::endl;
```

## 4.1 Runtime deduction (Superfluous chapter??)

The example above assumes we know which enum value we want to print at compile-time, but this is something we might not always know. So the following strategy is proposed:

1. We use `std::enum_string<T>(T t)` with some value, e.g. as input to a function.
2. The compiler detects this use and generates a lookup table (ie. `const char**`) inside the readonly section of our target binary.
3. At runtime, with little overhead, we make a lookup into this table and retrieves our string.

Creating such a table requires that each enum value has a certain *position* within its containing scoped enum, which we may query. This would necessitate the following interface:

```
namespace std {
  template<enum_type T> size_t enum_position(T t);
  template<enum_type> const char* enum_indexed_string(integral auto i);

  template<>
  struct enum_string<enum_type T> {
    const char* operator()(T t);
  };
}
```

The following code examplifies the usage:

```
void printColor(Color c) {
  std::cout << std::enum_string<Color>(c) << std::endl;
}

// Alternatively, if "std::cout" is provided with a meta-function overload:
void printColor(Color c) { std::cout << c << std::endl; }

// Testing:
printColor(Color::red); // Should output "<namespace::>Color::red"
```

# 5 Overview

This is the complete list of proposed functions, types, and meta-functions:

```
// header "<experimental/scoped_enum>" -- or -- "<experimental/enum>"

// Alternatively:
export module std.scoped_enum; // -- or --
export module std.enum;

namespace std {

  // Evaluates whether T is the name of a declared scoped enum type
  template<typename T>
  concept enum_type; // = /* implementation uncertain */

  // Checks if "V" is the underlying type for a declared scoped enum "T".
  template<enum_type T, is_integral_v V>
  concept enum_has_type = std::is_same_v<std::underlying_type_t<T>, V>;

  // A forward-iterator of all values inside a declared scoped enum
  template<enum_type T>
  struct enum_iterator;

  // A container with all values inside a declared scoped enum,
  // with methods `begin()`, `end()`, `cbegin()`, and `cend()` for
  // supporting forward iterating through the values by using the
  // "enum_iterator" defined above.
  template<enum_type T>
  struct enum_values;

  // Retrieves the underlying value represented by an enum value,
  // similar to "static_cast<underlying_type_t<T>".
  template<enum_type T, is_integral_v V>
  V enum_value(T t);

  template<enum_type T>
  bool enum_try_cast(integral auto v); // = /* implementation uncertain */

  // Retrieve the cardinality (ie. number of values) inside a declared
  // scoped enum.
  template<enum_type T>
  struct std::enum_cardinality { constexpr size_t value; };

  template<enum_type T>
  size_t std::enum_cardinality_v;

  // Retrieve a fully-qualified string representation of the provided
  // scoped enum value.
  template<enum_type T>
  const char* enum_string(T t); // = /* implementation compiler-specific */

  // Compare a scoped enum value with its underlying type (spaceship-operator)
  template<enum_type T, is_integral_v V>
  weak_ordering operator<=>(T lhs, V v);

  // Stream overload
  template<enum_type T>
```

```
  ostream& operator<<(ostream& stream, T t) {
    return stream << enum_string<T>(t);
  }

} // namespace std

// Typesafe cast from underlying value to declared scoped enum value,
// including a check for whether the provided value is valid.
template<std::enum_type T>
std::optional<T> enum_cast(std::integral auto v) {
  T out;
  if (std::enum_try_cast<T>(&out, v)) return out;
  else return std::nullopt;
}
```