C++ Scoped Enum proposal

Alexander Christensen

November 26, 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, the `typeinfo`-header 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 Substitution failure

Since the introduction of SFINAE, we have been empowered with manual template-substitution control. Because this proposal targets the modern C++ approach however, this proposal defines a concept instead. Similar to `typename` or `class`, we wish to perform a substitution when we know that a provided template-type is a scoped enum:

```
template<typename T>
concept std::enum_type = /* static check to verify if T is a scoped enum */

// Alternatively defined (shorter version) as
template<typename T>
concept std::enum;
```

We may then use this concept to constrain template-substitution:

```
template<std::enum T>
class MyClass { /* ... */ };
```

1.1 C++-23

Since C++23 adds a meta-function `std::is_scoped_enum`, it is evident that the language is already developing in the direction of this proposal. And as such, we would be able to trivially define our concept as such:

```
template<typename T>
concept std::enum = std::is_scoped_enum_v<T>;
```

Whether this addition to the language would be useful is up for discussion. Arguably, having a concept could potentially provide better compiler-output on substitution failure.

2 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<std::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 can maintain a dictionary of various enum members and bind them to a particular type "`V`".

3 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;
```

This is definitely better than the first example since we have replaced the `static_cast<std::underlying_type>` with a simpler interface. But we can do even better, especially with C++-20, if we provide an overload to the ”spaceship”-operator (three-way comparison), for comparing an enum value with its underlying type:

```
template<std::enum T>
std::weak_ordering operator<=>(T t, std::integral auto i);

// 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 can actually be used as a valid conversion. Thus, this document proposes an exception-safe `enum_cast`, similar to `dynamic_cast` but with the added benefit of being strictly *compile-time* evaluated:

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

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

4 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 Weekday { monday, tuesday /*, etc. */ };

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

5 String representation

Sometimes, for debugging purposes, we might want to be able to output to a terminal the string name of an enum value, possibly *fully-qualified* [1]. 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 };

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

printColor(Color::red); // Should output "Color::red"
```

[1] Including all containing namespaces, classes, structs, or modules.

5.1 Failed runtime deduction

If converting to scoped enum from an invalid underlying type, a string with the text `"<null>"` could be inserted. This would only happen if the conversion was not performed through the safe `enum_cast` but through an *unsafe* C-style cast (or an unchecked `static_cast`).

Let `"_a"` represent an integral to enum conversion which *might not* be valid, such as `"(Color)(rand())"`. The following piece of code should yield a compiler error, through either detectible non-`consteval` behaviour -or- compile-time detectible invalid conversion:

```
// This should fail compilation:
const char* str = std::enum_string<Color>(_a);
```

This would build upon the `enum_cast` returning a valid `optional<T>` or through constant propagation by the compiler. In either case, `enum_string_v` should be strictly *compile-time* evaluated. Suggested implementation is to generate a conversion table (`"const char**"`) inside the readonly section of the target binary. [2]

6 Enum position

In addition to a scoped enum's cardinality, this proposal also presents a function to get the *position* of an individual enum value inside its declaration. This is the proposed syntax:

```
enum class Fruit { banana = 1, apple = 2, orange = 4, clementine = 8 };
static_assert(std::enum_value_v<Fruit::orange> == 4);
static_assert(std::enum_position_v<Fruit::orange> == 2);
```

This, together with `enum_cardinality`, enables us to create constructions such as the following:

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

Since both `enum_cardinality` and `enum_position` are defined as compile-time constants, this code should never throw or index out-of-bounds.

[2]TODO: Would this be a good idea?

7 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;
}
```