C Scoped Enum Enhancements

ALEXANDER CHRISTENSEN

Contents

1 INTRODUCTION

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 }
```

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

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 funcionality, 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.

3 IMPACT ON THE STANDARD

This proposal requires new language features, because certain enum extensions cannot be covered by simply writing a library. The impacts are listed below with a subsection for each of them.

4 PROPOSED NOTATION

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.

5 CONVERT SCOPED ENUM TO STRING

Two ways of obtaining a string is suggested - the simple one, and a fully-qualified one which contains namespaces, classes, and other scopes:

```
1 namespace graphics {
2     enum class GraphicsApiType { none, opengl, vulkan };
3 }
4 std::cout << std::enum_str<GraphicsApiType::opengl>() << std::endl;
5 // --> "opengl"
6 std::cout << std::enum_str_full<GraphicsApiType::opengl>() << std::endl;
7 // "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`.

6 CONCEPT FOR "IS-AN-ENUM"

We can with C++-23 create a concept which is useful for template substitution:

```
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!
```

7 GET 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 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 is 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.

8 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{});
}
```

9 DESIGN DECISIONS

10 TECHNICAL SPECIFICATIONS

11 ACKNOWLEDGEMENTS

12 REFERENCES