Document number: P0974R0
Date: 2018-03-30
Audience: LEWG, LWG
Reply-to: Jonathan Ringle <jonringle at gmail dot com>

A Function Returning Whether An Underlying Type Value Is a Valid Enumerator of a Given Enumeration

Hello, Is It Me You're Looking For?

Table of Contents

I. Introduction

This proposal is to provide a safe way to determine if the value of an underlying type can be converted to a valid enumerator for a given enumeration type.

II. Motivation and Scope

There are situations, particularly when deserializing a stream of data, when a piece of data of an underlying type contains a value that should be assigned to a variable instance of an enum class type. However, since this data value may be coming from an unknown and/or untrusted source, it must be validated before trying to cast it to store it in the enum variable instance.

If all the enumerators in an enumeration is a set of sequential consecutive values, then a function to validate a value of the underlying type can be expressed as simply:

```
(Emin <= value <= Emax)
```

However, it is often the case that the list of enumerators in an enumeration is a non-sequential, non-consecutive list of values. In this case, a function to validate could be expressed in the following ways:

1. A switch statement on the value to test with a case for every valid enumerator that returns true, or returns false in the default case

2. A search for the value in a table of valid enumerators. The search can be optimized to a binary search if the table is maintained in a sorted order.

All of these options place an undue maintenance burden on the user (including the simple case where all of the enumerators in the enumeration are sequentially consecutive). An expression or function that performs an enum validation can be easily broken if new enumerator values are added or removed from the enumeration type, and the maintainer fails to also consider any validation code that may need to be updated as well.

Currently, there is a lack of support in C++ and the standard library to facilitate generalizing this particular kind of validation.

- There is no way to automatically determine the minimum and maximum enumerator values of an enumeration.
- There is no facility to automatically put into a container all of the valid enumerator values of an enumeration.
- There is no way to automatically determine the number of unique enumerator values of an enumeration (without resorting to preprocessor tricks).

If the maintainer wants to have validation code to validate that a underlying value is indeed a valid enumerator value of an enumeration, he/she must do so manually and separately from the enum definition using metadata that is derived and duplicated from the enum definition itself. Furthermore, if the user code needs similar validation code for other enum types, similar code must be duplicated and maintained separately.

There are implementations such as better-enums[1] that also tackle this problem (and go further to tackle other things such as enum conversion to/from strings). It does so by implementing reflection via preprocessor macros that then generate code to do its "magic".

The scope of this proposal is only concerned with defining a standard function interface that tests if an enumerator is valid for a given enumeration. There could be *future proposals* put forth to propose other things like an enum to string conversion as a template overload to the existing `std::to_string` interface:

```
namespace std {
template <typename E> requires is_enum<E>
std::string to_string(E value);
}
```

The following code implements the proposed `is_enumerator<E>` function using language and library facilities currently available to the user code in C++17. This implementation uses a traits pattern that *serves to help illustrate the shortcomings there currently are and what motivates this proposal by giving us a language in which to talk about the problems.*

The following traits are used:

- *`is_consecutive`*: a `bool` indicating if all the valid enumerators in the enumeration are sequentially consecutive
- *`min`*: the minimum enumerator value of type `E` that is found in the enumeration
- *`max`*: the maximum enumerator value of type `E` that is found in the enumeration
- *`is_sorted`*: a `bool` indicating if the enumerator values found in the `enumerators[]` trait are in a sorted order (it meets the sort requirements of the `std::binary_search` algorithm)
- *`count`*: a `size_t` holding the number of enumerators found in the enumeration. This is used to find the last element of the `enumerators[]` trait. This assumes that the number of elements in the `enumerators[]` traits is equal to the number of enumerators found in the enumeration. This assumption can be validated by user code with a `static_assert` and helps the maintainer with the burden of keeping up to date the `enumerators[]` trait.
- *`enumerators[]`*: an array holding an entry of type `E` for each valid enumerator found in the enumeration

Even though the below implementation of `is_enumerator<E>` consolidates in a single place the most efficient algorithm to perform the validation, it fails to ease any of the maintenance burdens that already exist. The following maintenance burdens are still inflicted upon the maintainer:

- If `is_consecutive=true`, then the maintainer has the burden to keep the value of the `min` and `max` traits up to date if there are enumerator values added or removed from the enumeration.
- If `is_consecutive=false`, the maintainer has the burden to keep the value of the `count` trait up to date if there are enumerator values added or removed from the enumeration.
- If `is_consecutive=false`, the maintainer has the burden to update the `enumerators[]` trait if there are enumerator values added or removed from the enumeration.
- If `is_consecutive=false` and `is_sorted=true`, then the maintainer has the additional burden to keep the `enumerators[]` trait in a sorted order if there are enumerator values added or removed from the enumeration. *The optimization benefit of `is_sorted=true` increases as the number of enumerator values in the enumeration increases. Unfortunately, this also increases the possibility of errors in the sort order due to the manual maintenance of the sorting.*

enum_cast.h:

```
#ifndef ENUM_CAST_H
#define ENUM_CAST_H

#include <algorithm>
#include <stdexcept>
#include <string>
#include <type_traits>

template <typename E>
struct enum_traits { };

template <typename E, size_t N>
constexpr const E* endof(const E ra[N]) { return ra + N; }

template <typename E>
bool is_enumerator(std::underlying_type_t<E> v)
{
  using traits = enum_traits<E>;
  using U = std::underlying_type_t<E>;
  if (traits::is_consecutive)
    return static_cast<U>(traits::min) <= v && v <= static_cast<U>(traits::max);

  constexpr const E* first = traits::enumerators;
  constexpr const E* last = endof<E, traits::count>(traits::enumerators);
  return (traits::is_sorted)
    ? std::binary_search(first, last, static_cast<E>(v))
    : std::find(first, last, static_cast<E>(v)) != last;
}

template <typename E>
E enum_cast(std::underlying_type_t<E> v)
{
  if (is_enumerator<E>(v))
    return static_cast<E>(v);
  using namespace std::literals;
  throw std::range_error("invalid enum value "s
                         + std::to_string(static_cast<int>(v))
                         + " for type "
                         + typeid(E).name());
}
#endif // ENUM_CAST_H
```

The below user code example defines the `enum class Crayola`.

The left side shows what the user code would need to do using the above implementation of `is_enumerator<E>` with C++17 and the additional code that the user must manually maintain. It employs a preprocessor “trick” to obtain a count of the number of valid enumerator values in the enumeration. Helpful comments are given to help maintainers of this code to not break the mechanism of the preprocessor “trick”. The resulting `ENUM_COUNT_` is stored in the `count` trait, which is then used in a `static_assert` (in `crayola.cc`) to verify that the number of valid enumerators in the enum class agrees with the number of elements found in the `enumerators[]` trait.

The right side shows what the user code would look like using the proposed template function `std::is_enumerator<E>` being made available to the user. The user no longer needs to try to determine the number of enumerators and no longer needs to maintain a separate table of valid enumerators.

Before	After

enum_cast.h:
 	
`#include <algorithm>`
`#include <type_traits>`
 	
`template <typename E>`
`struct enum_traits { };`
 	
`template <typename E, size_t N>`
`constexpr const E* endof(const E ra[N]) { return ra + N; }`
 	
`template <typename E>`
`bool is_enumerator(std::underlying_type_t<E> v)`
`{`
`  using traits = enum_traits<E>;`
`  using U = std::underlying_type_t<E>;`
`  if (traits::is_consecutive)`
`    return static_cast<U>(traits::min) <= v && v <= static_cast<U>(traits::max);`
 	
`  constexpr const E* first = traits::enumerators;`
`  constexpr const E* last = endof<E, traits::count>(traits::enumerators);`
`  return (traits::is_sorted)`
`    ? std::binary_search(first, last, static_cast<E>(v))`
`    : std::find(first, last, static_cast<E>(v)) != last;`
`}`	

crayola.h:
 	
`#include <cstdint>`
`#include <enum_cast.h>`
 	
`enum class Crayola : uint32_t {`
`  // Only one enumerator definition per line`
`  // No empty lines between BEGIN_COUNT_ and ENUM_COUNT_`
`  // No commented out enumerators between`
`  //    BEGIN_COUNT_ and ENUM_COUNT_`
`  BEGIN_COUNT_ = __LINE__,`
`  Red = 0xED0A3F,`
`  Maroon = 0xC32148,`
`  BrickRed = 0xC62D42,`
`  OliveGreen = 0xB5B35C,`
`  Asparagus = 0x7BA05B,`
`  Green = 0x3AA655,`
`  ForestGreen = 0x5FA777,`
`  TealBlue = 0x008080,`
`  Aquamarine = 0x95E0E8,`
`  SkyBlue = 0x76D7EA,`
`  Brown = 0xAF593E,`
`  ENUM_COUNT_ = __LINE__ - BEGIN_COUNT_ - 1,`
`};`
 	
`template <>`
`struct enum_traits<Crayola> {`
`  static constexpr const bool is_consecutive = false;`
`  static constexpr const Crayola min = Crayola{};// Doesn't matter when`
`  static constexpr const Crayola max = Crayola{};// is_consecutive=false`
`  static constexpr const bool is_sorted = false;`
`  static constexpr const size_t count =`
`      static_cast<size_t>(Crayola::ENUM_COUNT_);`
`  static const Crayola enumerators[];`
`};`
	crayola.h:
	
	`#include <cstdint>`
	
	`enum class Crayola : uint32_t {`
	`  Red = 0xED0A3F,`
	`  Maroon = 0xC32148,`
	`  BrickRed = 0xC62D42,`
	`  OliveGreen = 0xB5B35C,`
	`  Asparagus = 0x7BA05B,`
	`  Green = 0x3AA655,`
	`  ForestGreen = 0x5FA777,`
	`  TealBlue = 0x008080,`
	`  Aquamarine = 0x95E0E8,`
	`  SkyBlue = 0x76D7EA,`
	`  Brown = 0xAF593E,`
	`};`

crayola.cc:
 	
`#include "crayola.h"`
 	
`const Crayola enum_traits<Crayola>::enumerators[] = {`
`  Crayola::Red,`
`  Crayola::Maroon,`
`  Crayola::BrickRed,`
`  Crayola::OliveGreen,`
`  Crayola::Asparagus,`
`  Crayola::Green,`
`  Crayola::ForestGreen,`
`  Crayola::TealBlue,`
`  Crayola::Aquamarine,`
`  Crayola::SkyBlue,`
`  Crayola::Brown,`
`};`
 	
`template <typename T, size_t N>`
`constexpr size_t array_size(T (&)[N]) { return N; }`
 	
`static_assert(array_size(enum_traits<Crayola>::enumerators)`
`              == enum_traits<Crayola>::count,`
`              "Crayola enum mismatch");`	

crayola-usage.cc:

```
#include "crayola.h"

void crayola_usage()
{
  uint32_t underlying_col = 0xED0A3F;
  if (is_enumerator<Crayola>(underlying_col)) {
    // can be safely cast
    Crayola col = static_cast<Crayola>(underlying_col);
    std::cout << "color " << std::hex << static_cast<int>(col)
              << " is valid\n";
  } else {
    std::cout << "color " << std::hex << underlying_col
              << " is NOT valid\n";
  }
}
```

III. Impact On the Standard

Existing user code is not impacted by this addition, and provides an easy way for user code to validate an `underlying_type_t` value from an untrusted source without adding any additional maintenance burden beyond simply using the proposed `is_enumerator<E>` function to perform this validation.

The use of the `is_enumerator<E>` function in user code does potentially have an impact on memory usage since a table holding all valid enumerators in the enumeration *may* need to be created in order to perform this validation at runtime. However, if the user needed this functionality, and didn't use `is_enumerator<E>`, they would still need to implement something that has a similar cost. If the user has no need for enumeration validation for an enum type `E` that they may have in their code and doesn't invoke the `is_enumerator<E>` function in their code, there is no cost added to their code. This is a case of you pay for what you use.

IV. Design Decisions

This proposal is about specifying the interface signature for the `is_enumerator<E>` function and optionally for an `enum_cast<E>` conversion function. How these functions are implemented, whether it be as a compiler intrinsic function, or using static reflection[2] or some other means *is left as a standard library implementation detail*. This proposal should not be held dependent upon any future proposals (such as static reflection support that could be used to implement these functions). The `is_enumerator<E>` function is useful now and compatible with reflection. At the least, these functions could be implemented as compiler intrinsic functions now and converted to use a static reflection based implementation in the future without impacting user code.

Even though static reflection could give the tools to be able to implement the `is_enumerator<E>` function by user code, it should be provided by the standard library.

A possible implementation based on the Mirror reflection library[3] static reflection examples proposed in p0385r2[4] section 5.4 is shown below:

```cpp
template <typename E>
class enum_properties {
private:
    using namespace std;
    using U = underlying_type_t<E>;
    template <typename ... MEC>
    struct _hlpr {
        static void _eat(bool ...) { }
        static auto _make_map(void) {
            map<E, string> res;
            _eat(res.emplace(
                    reflect::get_constant_v<MEC>,
                    string(reflect::get_base_name<MEC>())
                    ).second...);
            return res;
        }
    };
    using ME = $reflect(E);
    using hlpr = reflect::unpack_sequence_t<
        reflect::get_enumerators_t<ME>,
        _hlpr
    >;
    auto m_ = hlpr::_make_map();
public:
    constexpr E min() const { return m_.begin()->first; }
    constexpr E max() const { return m_.rbegin()->first; }
    constexpr bool is_consecutive() const {
        U prev = static_cast<U>(m_.begin()->first);
        for (auto& e : m_) {
            if (prev == static_cast<U>(e.first)) continue;
            if (prev + 1 != static_cast<U>(e.first)) return false;
            prev = static_cast<U>(e.first);
        }
        return true;
    }
    constexpr bool has_enumerator(U v) {
        return m_.find(static_cast<E>(v)) != m_.end();
    }
    constexpr bool has_enumerator(E e) {
        return m_.find(e) != m_.end();
    }
    constexpr const string& to_string(E e) const {
        return m_.at(e);
    }
};


namespace std {
template <typename E>
constexpr bool is_enumerator(underlying_type_t<E> v)
{
    using U = underlying_type_t<E>;
    constexpr const enum_properties<E> enum_prop;
    if constexpr (enum_prop.is_consecutive())
        return static_cast<U>(enum_prop.min()) <= v && v <= static_cast<U>(enum_prop.max());
    else
        return enum_prop.has_enumerator(v);
}
template <typename E>
constexpr bool is_enumerator(E e)
{
    constexpr const enum_properties<E> enum_prop;
    return enum_prop.has_enumerator(e);
}
}
```

V. Technical Specifications

The proposed function should have the following signature and should return a `bool` indicating whether or not the passed in value of the `underlying_type_t<E>` argument is a valid enumerator found in the enumeration type `E`:

```
namespace std {
template <typename E> requires is_enum<E>
constexpr bool is_enumerator(underlying_type_t<E>);
}
```

It is possible to initialize an enum with an value that is not present as an enumerator value in the enumeration. Consider the following:

```
enum Color { Red = 1, Blue, Green };
Color color = Color{};
```

The variable `color` now holds an invalid value 0 that does not correspond to any enumerator in the enumeration. Therefore, it seems reasonable that someone might want to use `is_enumerator<E>` to validate if an enum variable of type `E` holds a valid enumerator. So the following overload should also be provided and should return a `bool` indicating whether or not the passed in value of enum type `E` argument itself holds a valid enumerator found in the enumeration type `E`:

```
namespace std {
template <typename E> requires is_enum<E>
constexpr bool is_enumerator(E);
}
```

Optional:

This also makes it possible to easily add an implementation of `enum_cast<E>` that is implemented in terms of `is_enumerator<E>`

```
namespace std {
template <typename E> requires is_enum<E>
E enum_cast(underlying_type_t<E> v)
{
    if (is_enumerator<E>(v))
        return static_cast<E>(v);
    throw std::range_error(...);
}
}
```

VI. Acknowledgements

I am grateful for the feedback received on:

https://groups.google.com/a/isocpp.org/forum/?utm_medium=email&utm_source=footer#!topic/std-proposals/41KbQVJMJL4

VII. References

[1]: Anton Bachin, better-enums, http://aantron.github.io/better-enums/
[2]: Matúš Chochlík, Axel Naumann, David Sankel, Static reflection, http://wg21.link/p0194r4
[3]: Matúš Chochlík, Mirror C++ reflection library documentation (current version)., http://matus-chochlik.github.io/mirror/doc/html/
[4]: Matúš Chochlík, Axel Naumann, David Sankel, Static reflection Rationale, design and evolution, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0385r2.pdf