C++ Scoped Enum proposal

Alexander Christensen

November 24, 2021

Following discussions on StackOverflow (¡insert links¿), it seems that there has been great interest in the subject of enums in C++. This proposal aims to being a discussion of improved *compile-time* support for enums, in particular enumeration and cast to underlying type.

For maintaining backwards-compatibility with C, this proposal *only* targets the scoped enums, identified by "`enum class <...>`".

0.1 Enumeration

Sometimes it may be desirable to enumerate through an enumeration 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, V>.begin();
  auto end = std::enum_iterator<T, V>.begin();
  for (auto it = begin; it != end; it++)  f.add_value(*it, V);

  // Alternatively:
  for (auto t : std::enum_values_v<T, V>)  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"`.

0.2 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 enum_value = MyEnum::Value;
bool is_zero = static_cast<std::underlying_type_t<T>>(enum_value) == 0;
```

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

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

```
namespace std {
  template<enum T, typename V>
  requires is_integral_v<V>
  optional<T> enum_cast(V v) {
    T out;
    if (enum_try_cast<T, V>(&out, v)) return out;
    else return nullopt;
  }
}
```

0.3 Overview

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

```
// header "experimental/scoped_enum"

// Alternatively:
export module std.scoped_enum;

namespace std {

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

  // A forward-iterator of all values inside a declared scoped enum
  template<typename T, is_integral_v V>
  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.
  template<typename T, is_integral_v V>
  struct enum_values;



}
```