

Enums

Montag, 11. September 2023 12:46

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

Intro

How would you add the logging facility (like with Decorator pattern) to the existing enum types in a generic way? The compiler limitation is C++17 standard.

@Disclaimer Once again, I don't want to use the already existing libraries out there that provide the desired feature out of box - but rather to find my own way to resolve this puzzle.

Implementation details

I've came up with two solutions, both somehow unsatisfactorily.

The first is the generic one, but not generic enough to stringify the enum names - from variadic pack of enumerators

We want to ensure, especially in embedded environment with C-style enums, as well with (C++) scoped enum classes - that the code behaves as expected.

That is why there are some helper - utility methods to perform these kind of checks: preferably at compile time

```
namespace details
{
    template <typename Enum>
    constexpr auto underlying_type(Enum e) noexcept
    {
        return static_cast<std::underlying_type_t<Enum>>(e);
    }

    template <typename T, typename E>
    constexpr bool is_same(const T val, const E e) noexcept
    {
        if constexpr (std::is_enum_v<E>) // E as enum class (scoped enums)
        {
            if constexpr (std::is_convertible_v<T, std::underlying_type_t<E>>) // val is underlying type
            {
                return val == underlying_type(e);
            }
            else if constexpr (std::is_same_v<T, E>) // val itself is enum
            {
                return val == e;
            }
        }
        else if constexpr (std::is_convertible_v<T, E>) // c-style enums (unscoped enums)
        {
            return val == e;
        }

        return false;
    }
} // namespace details
```

We either have the underlying value, or the enum instance itself, for which we want to find corresponding string representation

```
namespace details
{
    static constexpr std::size_t INVALID_INDEX = -1;

    // Find the enum instance that matches the given value
    // @see details::is_same() : the given value can be underlying enum value, or the
    // the enum instance itself (unscoped/scoped)
    template <typename T, typename E, std::size_t N>
    [[nodiscard]] constexpr std::size_t find(const T val, const std::array<E, N>& enums) noexcept
    {
        std::size_t index = INVALID_INDEX;

        for (std::size_t i = 0; i < N; ++i) {
            if (is_same(val, enums[i])) { index = i; break; }
        }

        return index;
    }
}
```

```

// Retrieve the enum instance from the underlying value, or default in case
// that there is no match
template <typename T, typename E, std::size_t N >
[[nodiscard]] constexpr decltype(auto) from_underlying_value(const T val,
    const std::array<E, N>& enums,
    const E defaultValue) noexcept
{
    const auto index = find(val, enums);
    return (index == INVALID_INDEX) ? defaultValue : enums[index];
}

// Enum instance (name) to the string representation, the same as Enum::toString() in Java
template<typename T, typename E, std::size_t N>
[[nodiscard]] constexpr decltype(auto) enum_to_string(const T val,
    const std::array<E, N>& enums,
    const std::array<std::string_view, N>& strings) noexcept
{
    const auto index = find(val, enums);
    return (index == INVALID_INDEX) ? "<n/a>": strings[index];
}
} // namespace details

```

I couldn't figure out better way of providing the stringify representation of enums name (like in Java Enum.toString()) than the following code

```

#define STRINGIFY(x) std::string_view(#x)

enum class AudioStream :std::uint8_t {main, alt, aux};

template <typename T>
[[nodiscard]] constexpr decltype(auto) audioStreamToString(T audioStream) noexcept
{
    using namespace details;

    return enum_to_string(
        audioStream, // this can be underlying value, enum itself (unscoped and scoped as well)
        createArray<AudioStream>(AudioStream::main, AudioStream::alt, AudioStream::aux),
        createArray<std::string_view> (
            STRINGIFY(AudioStream::main),
            STRINGIFY(AudioStream::alt),
            STRINGIFY(AudioStream::aux))
    );
}

```

This use the helper method details::createArray, for creating the std::array from variadic argument pack

```

template <typename T1, typename...Ts>
constexpr bool are_same_args = (std::is_same_v<T1, Ts> && ...); //fold expression

template <typename E, typename...Ts>
// requires is_same_args<E, Ts...> // C++20
constexpr auto createArray(Ts&&...args) noexcept
{
    static_assert(are_same_args<E, Ts...>, "The arguments type mismatch");
    // in C++23 the local static constexpr std::array can be created and return by the reference
    constexpr auto N = sizeof...(Ts);
    return std::array<E, N> {std::forward<Ts>(args)...};
}

```

The second approach is customized to the given enum type - it's not generic at all, it doesn't work with underlying types - but it can be convenient

```

#define ENUM_CHECK(X) case (X) : return STRINGIFY(X)

constexpr std::string_view printAudioStream(AudioStream audioStream) noexcept
{
    switch (audioStream)
    {
        ENUM_CHECK(AudioStream::main);
        ENUM_CHECK(AudioStream::alt);
        ENUM_CHECK(AudioStream::aux);
        default: break;
    }
    return "<n/a>";
}

```

Links

Compiler Explorer: <https://godbolt.org/z/ahEe36sbn>

<https://en.cppreference.com/w/cpp/language/enum>