Enums four ways

Wissam Mehio & Kris Jusiak

Motivation

```
1   enum StarWars{Zero, One, Two, Three};
2
3   const char* getName(StarWars val)
4   {
5       if (One == val )
6           return "The Phantom Menace";
7       else if (Two == val)
8           return "Attack of the Clones";
9       else if (Three == val)
10          return "Revenge of the Sith";
11      else return "Unsupported StarWars episode!!";
12  }
13
```

https://godbolt.org/g/zZYhMj

Method 1 – Boost Preprocessor

```cpp
#include <boost/preprocessor/variadic/to_seq.hpp>
#include <boost/preprocessor/seq/transform.hpp>
#include <boost/preprocessor/control/iif.hpp>
#include <boost/preprocessor/seq/enum.hpp>
#include <boost/preprocessor/tuple/elem.hpp>
#include <boost/vmd/is_tuple.hpp>
#include <array>
#include <string>
#include <string_view>
#include <type_traits>
#include <boost/preprocessor/arithmetic/add.hpp>
#include <boost/preprocessor/seq/for_each_i.hpp>


#define SOME_ENUM_CREATE_ELEMENT(r, d, i, enum_value)                                \
        BOOST_PP_TUPLE_ELEM(0, enum_value) = BOOST_PP_ADD(d,i),

#define SOME_MAPPING_STRINGS(r, d, enum_value)                                       \
std::string_view(BOOST_PP_TUPLE_ELEM(1, enum_value))

#define SOME_MAPPING(seq)                                                            \
        BOOST_PP_SEQ_ENUM(BOOST_PP_SEQ_TRANSFORM(SOME_MAPPING_STRINGS, _, seq))

#define SOME_ENUM_SEQ(name, seed, seq)                                               \
  namespace name {                                                                   \
        namespace {                                                                  \
        static constexpr std::array<std::string_view, BOOST_PP_SEQ_SIZE(seq)> data   \
         {{BOOST_PP_SEQ_ENUM(BOOST_PP_SEQ_TRANSFORM(SOME_MAPPING_STRINGS, _, seq))   \
         }};}                                                                       \
        enum  name_enum {                                                            \
                BOOST_PP_SEQ_FOR_EACH_I(SOME_ENUM_CREATE_ELEMENT, seed, seq)         \
                name_enum_end                                                        \
        };                                                                           \
        static constexpr auto getString(name_enum enum_val) {                       \
                        return data[static_cast<std::size_t>(enum_val)-seed];}       \
  };

#define SOME_ENUM(name, seed, ...)                                                   \
  SOME_ENUM_SEQ(name, seed, BOOST_PP_VARIADIC_TO_SEQ(__VA_ARGS__))
```

```cpp
SOME_ENUM(StarWars, 99,
                (One, "The Phantom Menace"),
                (Two, "Attack of the Clones")
                );
int main()
{
        static_assert(StarWars::getString(StarWars::One) == "The Phantom Menace");
        static_assert(StarWars::getString(StarWars::Two) == "Attack of the Clones");
        static_assert(StarWars::One == 99);
        static_assert(StarWars::Two == 100);
        return 0;
}
```

- *Quicknir –WiseEnum*
- *Anton Bachin – BetterEnums*

- *Contiguous numbers*
- *Have to create DSL for default values*
- *Namespace introduces problems*

https://godbolt.org/g/upQUHd

Method 2 – Pretty_Function

```cpp
#include <iostream>
#include <string>
#include <string_view>

template <typename ENUM_TYPE, ENUM_TYPE ENUM_VALUE>
constexpr std::string_view EnumName()
{
   const char* abc = __PRETTY_FUNCTION__ + 64;
   std::string_view sv (abc);
   std::string_view ret {sv.data(), sv.length()-1};
   return ret;
}

template <typename ENUM_TYPE, ENUM_TYPE ENUM_COUNT, ENUM_TYPE ENUM_VALUE>
struct EnumMatch
{
  constexpr static std::string_view Do(ENUM_TYPE enum_value)
  {
    if (enum_value == ENUM_VALUE)
      return EnumName<ENUM_TYPE, ENUM_VALUE>();
    return EnumMatch<ENUM_TYPE, ENUM_COUNT, ENUM_TYPE(ENUM_VALUE+1)>::Do(enum_value);
  }
};

template <typename ENUM_TYPE, ENUM_TYPE ENUM_COUNT>
struct EnumMatch<ENUM_TYPE, ENUM_COUNT, ENUM_COUNT>
{
   constexpr static std::string_view Do(ENUM_TYPE )
   {
     return "Enum not found";
   }
};
#define ENUM_NAME(enum_type, enum_value) \
    EnumMatch<enum_type, enum_type##_Count, enum_type(0)>::Do(enum_value)

```

```cpp
enum TestEnum
{
  Zero, One, Two, Three, Four, Five, Six, Seven, Eight, Monkeys,
  TestEnum_Count
};

int main()
{
    static_assert(ENUM_NAME(TestEnum, Zero) == "Zero");
    static_assert(ENUM_NAME(TestEnum, Monkeys) == "Monkeys");
    return 0;
}
```

- *TestEnum_Count*
- *Namespace dependent*
- *Can't create custom strings*
- *Compiler specific*

https://godbolt.org/g/UXh87v

Method 3 – Values as types

```cpp
#include <utility>
#include <string_view>
#include <cmath>

template<char... Cs>
struct string {
  constexpr operator std::string_view() const {
    constexpr char name[] = { Cs..., 0 };
    return name;
  }
};

template <class T, T... Chrs>
constexpr auto operator""_s() { return string<Chrs...>{}; }

template <char... Cs>
constexpr auto operator""_c() {
  return []<auto... Ns>(std::index_sequence<Ns...>) {
    return std::integral_constant<int, ((int(std::pow(10, sizeof...(Ns) - Ns - 1)) * (Cs - '0')) + ...)>{};
  }(std::make_index_sequence<sizeof...(Cs)>{});
}

template<class N, class TName = string<>>
struct value {
   constexpr value(N, TName = string<>{}) {}
   constexpr explicit operator int() const { return N{}; }
   constexpr auto c_str() const { return std::string_view{TName{}}; }
};

```

```cpp
struct StarWars {
  static constexpr auto One = value{1_c, "The Phantom Menace"_s};
  static constexpr auto Eleven = value{11_c};
};

static_assert(1 == sizeof(StarWars));
static_assert(1 == static_cast<int>(StarWars::One));
static_assert(11 == static_cast<int>(StarWars::Eleven));

static_assert("The Phantom Menace" == StarWars::One.c_str());
static_assert("" == StarWars::Eleven.c_str());

```

- *Not really an enum*
- *No auto increment*

https://*godbolt.org/g/xfSmnu*

Method 4 – Stateful Metaprogramming

```cpp
#include <utility>
#include <string_view>

template <int N>
struct flag{
    friend constexpr int adl_flag(flag<N>);
};

template <int N>
struct writer{
    friend constexpr int adl_flag(flag<N>){return N;}
    constexpr static int value = N;
};

template <int N, int = adl_flag(flag<N>{})>
int constexpr reader(int, flag<N>){
    return N;
}

template <int N>
int constexpr reader (float, flag<N>, int R = reader(0, flag<N-1>{}) ){return R;}

int constexpr reader (float, flag<0>){
    return 0;
}

template <class RetT, int N=1>
RetT constexpr getNext(int R = writer<reader(0, flag<256>{}) + N>::value){return R;}
```

```cpp
template<class EnumBaseT = int, class EnumDetailsT = std::string_view>
struct StarWars_{
  constexpr static inline auto value1 =
            value<EnumBaseT, EnumDetailsT>{getNext<EnumBaseT>(), "The Phantom Menace"};
  constexpr static inline auto value2 =
            value<EnumBaseT, EnumDetailsT>{getNext<EnumBaseT>(), "Attack of the Clones"};
};

int main()
{
    using StarWars = StarWars_<uint8_t>;
    static_assert(StarWars::value1.id == 1);
    static_assert(StarWars::value2.id == 2);
    static_assert(StarWars::value1.detail == "The Phantom Menace");
    static_assert(StarWars::value2.detail == "Attack of the Clones");
}
```

- *Filip Roséen – constexpr counter*

- *Not really an enum*
- *Auto inc = can not set values*
- *Filip Roséen – constexpr counter*

https://godbolt.org/g/iWpuZw

References

- https://github.com/quicknir/wise_enum
- http://aantron.github.io/better-enums/
- http://b.atch.se/posts/constexpr-counter/