Enhanced Enum

Jaakko Moisio

Feb 17, 2022

CONTENTS:

CHAPTER

ONE

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

TWO

INTRODUCTION

Enhanced Enum is a library that gives C++ enums capabilities that they don't normally have:

```
enum class StatusLabel {
    INITIALIZING,
    WAITING_FOR_INPUT,
    BUSY,
};

constexpr auto status = Statuses::INITIALIZING;
```

Their value is no longer restricted to integers:

```
static_assert( status.value() == "initializing" );
static_assert( status == Status::from("initializing") );
```

They can be iterated:

```
std::cout << "Listing " << Statuses::size() << " enumerators:\n";
for (const auto status : Statuses::all()) {
    std::cout << status.value() << "\n";
}
```

...all while taking remaining largely compatible with the fundamental enums:

```
static_assert( sizeof(status) == sizeof(StatusLabel) );
static_assert( status == StatusLabel::INITIALIZING );
static_assert( status != StatusLabel::WAITING_FOR_INPUT );
```

2.1 Why yet another enum library for C++?

There are plethora of options available for application writers that want similar capabilities than this library provides. Why write another instead of picking one of them?

Short answer: Because it solved a problem for me, and I hope it will solve similar problems for other people

Longer answer: There is a fundamental limitations to the capabilities of native enums within the standard C++, and in order to cope with them, enum library writers must choose from more or less unsatisfactory options:

- Resort to compiler implementation details. While this is a non-intrusive way to introduce reflection, it's not what I'm after.

- Use macros. By far the most common approach across the ecosystem is to use preprocessor macros to generate the type definitions. To me macros are just another form of code generation. The advantage is that this approach needs standard C++ compiler only. The drawback is the inflexibility of macro expansions.

Enhanced Enum utilizes a proper code generator to create the necessary boilerplate for enum types. The generator is written in Python, and unlocks all the power and nice syntax that Python provides. The generated code is clean and IDE friendly. This approach enables the enums created using the library to have arbitrary values, not just strings derived from the enumerator names. The drawback is the need to include another library in the build toolchain.

2.2 Getting started

The C++ library is header only. Just copy the contents of the `cxx/include/` directory in the repository to your include path. If you prefer, you can create and install the CMake targets for the library:

```
$ cmake /path/to/repository
$ make && make install
```

In your project:

```
find_package(EnhancedEnum)
target_link_libraries(my-target EnhancedEnum::EnhancedEnum)
```

The enum definitions are created with the EnumECG library written in Python. It can be installed using `pip`:

```
$ pip install EnumECG
```

The library and code generation API are documented in the user guide hosted at Read the Docs.

2.3 Contact

The author of the library is Jaakko Moisio. For feedback and suggestions, please contact jaakko@moisio.fi.

CHAPTER

THREE

USER GUIDE

Enhanced Enum is a header-only C++ library used to implement the enhanced enumeration types. EnumECG is a Python library is used to generate the enhanced enum definitions for the C++ application.

3.1 Enhanced Enum – The guide

3.1.1 Motivation

The native C++ enums are a good choice for types labeling choices from a limited set. They are

- Type safe: It's hard for a programmer to accidentally create an enum object holding a value not in the predetermined set.
- Lightweight: Under the hood enum is just an integer

They also have restrictions which sometimes makes them tedious to work with:

- Enum is just an integer. The concept an enum is modeling may well have a more natural representation, but with a native C++ enum that mapping is not implemented in the enum type itself.
- They lack reflection support. Even iterating over (or should I say *enumerating*) the members of an enum type requires writing boilerplate and duplicating enumerator definitions.

This library attempts to solve those problems by helping creating *enhanced enum* types that are just as lightweight and type safe as native enums, but support the convenient features users of higher level languages have learned to expect from their enums.

The design goals are:

- **Standard compliant:** The library doesn't use compiler specific features to enable reflection capabilities.
- **IDE friendly:** The library doesn't use macros to generate the unavoidable boilerplate. The generated code is explicit and available for human and tool inspection.
- **Supporting modern C++ idioms:** The library is `constexpr` correct, includes utilities for template programming and type checks, etc.
- **Zero-cost:** Ideally code manipulating enhanced enums should compile down to the same instructions as code manipulating native enums.

To give the enum types their capabilities without resorting to compiler hacks, it's necessary to write some boilerplate accompanying the enum definitions. To aid with that the project includes `EnumECG` library that can be used to generate the necessary C++ code with Python. See *EnumECG – The code generation support* for more details.

3.1.2 Creating the enumeration

Warning: The generated code should not be edited, or the behavior of instantiating and using a class deriving from enum_base is undefined. The library makes assumptions about the types and functions used with the library. Those assumptions include but are not limited to:

- The values of the label enumerators are zero-based integer sequence that can be used as indices in the `values` array.
- The enhanced enum instantiates enum_base with correct template arguments, and has no non-static data members or non-empty bases.

As discussed if *Motivation*, some boilerplate is needed to define an enhanced enum type. The library tries to keep the generated boilerplate minimal, clean, and part of API. This means that the user of the generated enum type, and not just the library machinery, is free to use the types, constants and functions that the. Because the generated definitions are also public API of the type, backward incompatible changes are not made lightly.

Let's create `Status` type that enumerates the different states of an imaginary process. The boilerplate can be generated with the EnumECG library, described in detail in *EnumECG – The code generation support*.

```
>>> import enum
>>> class Status(enum.Enum):
...     INITIALIZING = "initializing"
...     WAITING_FOR_INPUT = "waitingForInput"
...     BUSY = "busy"
>>> import enumecg
>>> enumecg.generate(Status)
'...'
```

The above command will generate the following C++ code:

```
enum class StatusLabel {
    INITIALIZING,
    WAITING_FOR_INPUT,
    BUSY,
};

struct EnhancedStatus : ::enhanced_enum::enum_base<EnhancedStatus, StatusLabel,␣
↪std::string_view> {
    using ::enhanced_enum::enum_base<EnhancedStatus, StatusLabel, std::string_view>
↪::enum_base;
    static constexpr std::array values {
        value_type { "initializing" },
        value_type { "waitingForInput" },
        value_type { "busy" },
    };
};

constexpr EnhancedStatus enhance(StatusLabel e) noexcept
{
    return e;
}

```

(continues on next page)

(continued from previous page)

```
21 namespace Statuses {
22 inline constexpr const EnhancedStatus::value_type& INITIALIZING_VALUE { std::get<0>
   ↪(EnhancedStatus::values) };
23 inline constexpr const EnhancedStatus::value_type& WAITING_FOR_INPUT_VALUE { std::get<1>
   ↪(EnhancedStatus::values) };
24 inline constexpr const EnhancedStatus::value_type& BUSY_VALUE { std::get<2>
   ↪(EnhancedStatus::values) };
25 inline constexpr EnhancedStatus INITIALIZING { StatusLabel::INITIALIZING };
26 inline constexpr EnhancedStatus WAITING_FOR_INPUT { StatusLabel::WAITING_FOR_INPUT };
27 inline constexpr EnhancedStatus BUSY { StatusLabel::BUSY };
28 inline constexpr auto begin() noexcept { return EnhancedStatus::begin();  }
29 inline constexpr auto end() noexcept { return EnhancedStatus::end();  }
30 inline constexpr auto all() noexcept { return EnhancedStatus::all();  }
31 }
```

The generated boilerplate may appear in a namespace scope (global or any other namespace) in the user's C++ files. In addition the file must include the `enhanced_enum.hh` header file.

Overview of the generated code

The code starts with the definition of `enum class StatusLabel` at line 1. This is the underlying *label enum* type. The *label enumerators* be thought as a names for the enumerators in the enhanced enum type.

The next block is the definition of `struct EnhancedStatus` at line 7. This is the actual enhance enum type. It derives from `enum_base` implemented in the Enhanced Enum library header. The base class has three template arguments:

1. `EnhancedStatus` to employ the curiously recurring template pattern.
2. `StatusLabel`, the label enum type
3. `std::string_view`, the value type of the enumerators. They are discussed in more detail in *Enumerator types and values*.

The class also defines static data members mapping the enumerators to their values.

The library needs a way to map a label enumerators to the corresponding enhanced enumerators without knowing the name of the enhanced enum type. That is done with the `enhance()` method, defined at line 16. It needs to be defined in the same namespace as `StatusLabel` itself to support argument-dependent lookup. Because the library needs to reserve an identifier in the user namespace, there is a risk for name collision. The name `enhance` was chosen because, although short, it is a verb not otherwise often used in computer programming. Due to its shortness it makes the code using the function cleaner.

Finally the enumerators and their values are defined as constants in the `namespace Statuses`, defined at line 21. This *associate namespace* is not necessary for the library itself, but the application may use the constants and functions in this namespace for convenience.

Controlling the output

The above command generated names of the types, enumerators and the helper namespace from the names in the `class Status`. The defaults may not be what you want. Especially you might want to control if the label enum or the enhanced enum is the *primary type* and simply called `Status`.

For the details about controlling output see *Enum definitions in detail*.

Integrating to a project

The Enhanced Enum library does not provide integration between the `EnumECG` and the development environment out of box. It is recommended to use template based code generator tooling to include the bits generated by `EnumECG` to your C++ header files.

Here is an example using the awesome cog library. Write a header file like the following:

```
#include <enhanced_enum/enhanced_enum.hh>

#include <string_view>

namespace myapp {

/*[[[cog
import cog
import enumecg
import enum
class Status(enum.Enum):
    INITIALIZING = "initializing"
    WAITING_FOR_INPUT = "waitingForInput"
    BUSY = "busy"
cog.out(enumecg.generate(Status))
]]]*/
//[[[end]]]

}
```

Then, assuming you have `cog` installed in your environment, just invoke the command line utility and the enum boilerplate will appear where the template is located in the source file:

```
$ cog -r status.hh
```

`cog` supports both in-place code generation and writing the output to a file. There are advantages and disadvantages in both approaches. In-place code generation is IDE friendly and allows users that don't have `cog` or `EnumECG` installed still compile your code, but care must be taken that the generated code is not changed manually. See the `cog` documentation for more details.

3.1.3 Using the enumeration

This section introduces how to use an enhanced enum type in your code, and the basic properties of an enhanced enum type. It is assumed that the code is the one generated in the previous section (*Creating the enumeration*).

- Label type called `StatusLabel`
- Enhanced enum type called `EnhancedStatus`
- Function `enhance()` that can be used to promote a label enum into enhanced enum
- Associate namespace `Statuses` containing enums and their values as constants

Basic properties

Enhanced enums, like the build-in C++ enums, are regular. They can be constructed and assigned from enhanced and label enums:

```
auto status = enhance(StatusLabel::INITIALIZING);
assert( status.get() == StatusLabel::INITIALIZING );
status = StatusLabel::BUSY;
assert( status.get() == StatusLabel::BUSY );
```

They have all comparison operators defined, and working transparently with both enhanced and label enum operands. Both the enhanced enums and label enums are totally ordered by the order the labels are declared in the code.

```
static_assert( Statuses::INITIALIZING == StatusLabel::INITIALIZING );
static_assert( StatusLabel::INITIALIZING < Statuses::BUSY );
// etc...
```

In C++20 the three-way comparison operator is defined between both label and enhanced enums.

```
static_assert( (Statuses::INITIALIZING <=> StatusLabel::INITIALIZING) == std::strong_
↪ordering::equal );
static_assert( (StatusLabel::INITIALIZING <=> Statuses::BUSY) == std::strong_
↪ordering::less );
```

Enumerator values and labels

Enhanced enumerators have values. They can be accessed using the `value()` function

```
static_assert( Statuses::INITIALIZING.value() == "initializing" );
```

Enumerators can be constructed from value using the static `from()` method:

```
static_assert( EnhancedStatus::from("initializing") == Statuses::INITIALIZING );
```

The underlying label enum can be accessed either with the `get()` method or explicit cast. Note that although label enum is implicitly convertible to enhanced enum, the converse is deliberately explicit.

```
static_assert( Statuses::INITIALIZING.get() == StatusLabel::INITIALIZING )
static_assert( static_cast<StatusLabel>(Statuses::INITIALIZING) ==␣
↪StatusLabel::INITIALIZING );
```

Enumerator ranges

The number of enumerators in an enhanced enum type can be queries by using the `size()` and `ssize()`, for unsigned and signed sizes, respectively.

A range containing all enumerators of a given enum type can be constructed with the static `all()` method:

```
for (const auto status : EnhancedStatus::all()) {
    // use status
}
```

The returned range can be used in compile time and has all the enumerators in the same order as they are declared in the type.

In C++20 the range returned by `all()` models a random access view, and can be used with other utilities of the ranges library.

```
const status_values_reversed =  EnhancedStatus::all() |
    std::ranges::views::reverse |
    std::ranges::views::transform([](const auto e) { return e.value(); });
```

For interfaces consuming iterator pairs, using `begin()` and `end()` may be more convenient:

```
std::for_each(
    EnhancedStatus::begin(), EnhancedStatus::end(),
    [](const auto status) { /* use status */ });
```

Note: The user should not assume an underlying type returned by the `all()`, `begin()` and `end()` functions, except that the iterators supports random access.

The iterators model the C++17 random access iterator concepts. The range returned by `all()` *doesn't* model STL container. The intention is to remain forward-compatible with the view concepts from the Ranges TS. Unlike STL containers, views don't define type aliases etc.

In C++20 the enumerator ranges implement `std::ranges::view_interface`, and are thus compatible with native views.

For convenience and readability, the associate namespace generated by the `EnumECG` library also contains aliases to the range functions:

```
for (const auto status : Statuses::all()) {
    // use status
}

std::for_each(
    Statuses::begin(), Statuses::end(),
    [](const auto status) { /* use status */ });
```

Hashes

The library defines a hash template for enhanced enumerations:

```
enhanced_enum::hash<EnhancedStatus> hasher;
for (const auto status : EnhancedStatus::all()) {
    std::cout << "Hash of " << status.value() << " is " << hasher(status) << "\n";
}
```

Due to restrictions of specializing the `std::hash` template in the standard library[1], the hash object is defined in the `enhanced_enum` namespace. But the standard library by default expects to find the definition of a hash object in the `std` namespace. You can do this yourself via inheritance:

```
namespace std {
    template<>
    struct hash<EnhancedStatus> : enhanced_enum::hash<EnhancedStatus> {};
}

std::unordered_map<EnhancedStatus, int> map;
map[Statuses::INITIALIZING] = 123;
```

Alternatively you can just instantiate the hash template explicitly when needed:

```
std::unordered_map<EnhancedStatus, int, enhanced_enum::hash<EnhancedStatus>> map;
map[Statuses::INITIALIZING] = 123;
```

3.1.4 Library reference

The base class

template<typename **EnhancedEnum**, typename **LabelEnum**, typename **ValueType**>

struct `enhanced_enum::`**`enum_base`**

Base class for the enhanced enumeration types.

The essential functionality of an enhanced enum type is implemented in this class. A type gains the capabilities of an enhanced enum by deriving from this class, and defining a static constant array called `values`, as described in the user guide. The code for the derived class is intended to be autogenerated to ensure conformance with the requirements.

There is an order-preserving isomorphism between the instances of `EnhancedEnum`, and the enumerators of the underlying `LabelEnum`. Thus, conceptually the instances of `EnhancedEnum` are its enumerators, and will be called so in the documentation.

An enumerator of `EnhancedEnum` simply stores the value of its label enumerator as its only member. Consequently, most of the traits of `LabelEnum` are also traits of the `EnhancedEnum` type: enhanced enumerations are regular, totally ordered, trivial, and layout compatible with their underlying label enumerators.

It is assumed that the definition of the derived class is compatible with the requirements imposed by the library. Most notably the derived class must not have any non-static data members or other non-empty base classes. The intended way to ensure compatibility is to autogenerate the definition.

[1] Because the Enhanced Enum library doesn't know about *your* types, the C++17 implementation relies on SFINAE to specialize templates for enhanced enumerations. But the standard library doesn't allow SFINAE, it only allows partial specializations in the `std` namespace to rely on user defined types explicitly.

Warning: Trying to use an instance of a class derived from *enum_base* not meeting the requirements imposed by the library results in undefined behavior.

Template Parameters

- **EnhancedEnum** – The enhanced enumeration. The base class template uses curiously recurring template pattern.
- **LabelEnum** – The underlying enumeration (`enum class`)
- **ValueType** – The enumerator value type

Public Types

using **label_type** = *LabelEnum*
Alias for the label enum type.

using **value_type** = *ValueType*
Alias for the value type.

Public Functions

enum_base() = default
Default constructor.

Construct an enumerator with indeterminate value

constexpr **enum_base**(const *enum_base* &other) noexcept = default
Copy construct an enumerator.

Postcondition: `this->get() == other.get()`

Parameters **other** – The source enumerator

inline constexpr **enum_base**(const *label_type* &label) noexcept
Construct an enumerator with the given label.

Postcondition: `this->get() == label`.

Parameters **label** – The label enumerator

constexpr *enum_base* &**operator=**(const *enum_base* &other) noexcept = default
Copy assign an enumerator.

Postcondition: `this->get() == other.get()`

Parameters **other** – The source enumerator

Returns `*this`

inline constexpr *label_type* **get**() const noexcept
Return the label enumerator.

inline explicit constexpr **operator** *label_type*() const noexcept
Convert the enumerator to its underlying label enumerator.

Returns `this->get()`

inline constexpr const *value_type* &**value**() const noexcept
Return the value of the enumerator.

Throws `std::out_of_range` – if `*this` is not a valid enumerator

Public Static Functions

static inline constexpr std::size_t **size**() noexcept
Return the size of the enumeration.

static inline constexpr std::ptrdiff_t **ssize**() noexcept
Return the signed size of the enumeration.

static inline constexpr auto **begin**() noexcept
Return iterator to the first enumerator.

Returns A random access iterator to the beginning of the range containing all enumerators in the order they are declared in the enum

static inline constexpr auto **end**() noexcept
Return iterator to one past the last enumerator.

Returns Iterator to the end of the range pointed to by *begin()*

static inline constexpr auto **all**() noexcept
Return range over all enumerators.

Returns A range containing all enumerators in the order they are declared in the enum

static inline constexpr std::optional<*EnhancedEnum*> **from**(const *value_type* &value) noexcept
Return the enumerator with the given value.

Note: The number of comparisons is linear in the size of the enumeration. The assumption is that the number of enumerators is small and the values are localized in memory, making linear algorithm efficient in practice.

Parameters **value** – The value to search

Returns The first enumerator whose value is `value`, or empty if no such enumerator exists

Template support

group **templatesupport**
Support for writing templates with label enumerations and enhanced enumerations.

Typedefs

using **enhanced** = decltype(enhance(std::declval<LabelEnum>()))
Convert a label enumeration to an enhanced enumeration.

Template Parameters **LabelEnum** – A label enum type (`enum class`)

using **make_enhanced_t** = typename make_enhanced<Enum>::type
Shorthand for *make_enhanced*.

Functions

template<typename **Enum**>
constexpr *make_enhanced_t*<*Enum*> **ensure_enhanced**(*Enum* e) noexcept
Return enhanced enumerator associated with the argument.

See also:

is_same_when_enhanced for an example how this might be used in generic code

Parameters **e** – The enumerator to promote

Returns If `Enum` is a label enumeration, promote `e` to the associated enhanced enumerator. If `Enum` is an enhanced enumeration, return `e` as is.

Variables

template<typename **T**>

constexpr bool **is_enhanced_enum_v** = is_enhanced_enum<*T*>::value
Shorthand for *is_enhanced_enum*.

template<typename **T**>

constexpr bool **is_label_enum_v** = is_label_enum<*T*>::value
Shorthand for *is_label_enum*.

template<typename **T**, typename **U**>

constexpr bool **is_same_when_enhanced_v** = is_same_when_enhanced<*T*, *U*>::value
Shorthand for *is_same_when_enhanced*.

template<typename **T**>

struct **is_enhanced_enum**
#include <enhanced_enum.hh> Check if a type in an enhanced enumeration.

If `T` is a type that derives from *enum_base*, then `is_enhanced_enum` derives from `std::true_type`. Otherwise derives from `std::false_type`.

Template Parameters **T** – The type to check

template<typename **T**>

struct **is_label_enum**
#include <enhanced_enum.hh> Check if a type in a label enumeration.

If T is a type that has an associated enhanced enum type (see *enhanced*), the *is_label_enum* derives from std::true_type. Otherwise derives from std::false_type.

Template Parameters T – The type to check

template<typename **Enum**>

struct **make_enhanced**

#include <enhanced_enum.hh> Makes an enumeration enhanced.

If Enum is either an enhanced enum (see *is_enhanced_enum*) or a label enum (see *is_label_enum*), has the associated enhanced enum as member typedef type. Otherwise has no member typedefs.

template<typename **T**, typename **U**>

struct **is_same_when_enhanced**

#include <enhanced_enum.hh> Check if two types are the same once enhanced.

If T and U are either label enums or enhanced enums, and the associated enhanced enum types are the same, derives from std::true_type. Otherwise derives from std::false_type.

This template is useful for writing generic comparison functions that accepts both label and enhanced enums of the same kind.

```
template<
    typename Enum1, typename Enum2,
    typename = std::enable_if_t<is_same_when_enhanced_v<Enum1, Enum2>>
>
bool compare(Enum1 e1, Enum2 e2)
{
    // implementation can use ensure_enum(e1) to access the enhanced␣
↪capabilities
}
```

Hash

template<typename **Enum**>

struct **hash**

Hash for enhanced enums.

This is a struct template satisfying the requirements of Hash for enhanced enum types. Instantiating the template is possibly if and only if *is_enhanced_enum_v<Enum>* == true.

Note: Due to restrictions of the C++ standard library, the enhanced enum library doesn't specialize the std::hash template. Please see the user guide if you need a specialization of std::hash for your own type.

3.2 EnumECG – The code generation support

3.2.1 Overview

EnumECG (almost acronym for Enhanced Enum Code Generator) is a Python library accompanying the Enhanced Enum library. It is used to generate C++ boilerplate for the enhanced enum types. Please see *Enhanced Enum – The guide* for more information about the design of the library.

There are multiple ways to map a Python object to the C++ enum type. The following code examples all produce the same C++ boilerplate. For further discussion see *Creating the enumeration*.

```
enum class StatusLabel {
    INITIALIZING,
    WAITING_FOR_INPUT,
    BUSY,
};

struct EnhancedStatus : ::enhanced_enum::enum_base<EnhancedStatus, StatusLabel,␣
↪std::string_view> {
    using ::enhanced_enum::enum_base<EnhancedStatus, StatusLabel, std::string_view>
↪::enum_base;
    static constexpr std::array values {
        value_type { "initializing" },
        value_type { "waitingForInput" },
        value_type { "busy" },
    };
};

constexpr EnhancedStatus enhance(StatusLabel e) noexcept
{
    return e;
}

namespace Statuses {
inline constexpr const EnhancedStatus::value_type& INITIALIZING_VALUE { std::get<0>
↪(EnhancedStatus::values) };
inline constexpr const EnhancedStatus::value_type& WAITING_FOR_INPUT_VALUE { std::get<1>
↪(EnhancedStatus::values) };
inline constexpr const EnhancedStatus::value_type& BUSY_VALUE { std::get<2>
↪(EnhancedStatus::values) };
inline constexpr EnhancedStatus INITIALIZING { StatusLabel::INITIALIZING };
inline constexpr EnhancedStatus WAITING_FOR_INPUT { StatusLabel::WAITING_FOR_INPUT };
inline constexpr EnhancedStatus BUSY { StatusLabel::BUSY };
inline constexpr auto begin() noexcept { return EnhancedStatus::begin();  }
inline constexpr auto end() noexcept { return EnhancedStatus::end();  }
inline constexpr auto all() noexcept { return EnhancedStatus::all();  }
}
```

Creating C++ enum from Python enum

The most idiomatic way to create an enhanced enum type is to give the generator a Python enum type.

```
>>> import enum
>>> class Status(enum.Enum):
...     INITIALIZING = "initializing"
...     WAITING_FOR_INPUT = "waitingForInput"
...     BUSY = "busy"
>>> import enumecg
>>> enumecg.generate(Status)
'...'
```

The mapping between the name of the enum type, and the names and values of the enum members are obvious in this style.

Creating C++ enum from a mapping

This is a convenient method if the enum definition is loaded from a file using general purpose serialization format like JSON or YAML.

```
>>> status = {
...     "typename": "Status",
...     "members": [
...         {
...             "name": "INITIALIZING",
...             "value": "initializing",
...         },
...         {
...             "name": "WAITING_FOR_INPUT",
...             "value": "waitingForInput",
...         },
...         {
...             "name": "BUSY",
...             "value": "busy",
...         },
...     ]
... }
>>> import enumecg
>>> enumecg.generate(status)
'...'
```

The supported keys are:

- `typename`: The enum typename.
- `members`: Mapping between enumerator names and values.
- `docstring`: An optional documentation accompanying the generated types, constants and functions. See *Including documentation in the generator output* for details.

Native representation

The code generator uses *enumecg.definitions.EnumDefinition* as its datatype holding the native representation of an enum definition. They can be used with the generator directly if a very fine control of the generated code is required.

```
>>> from enumecg.definitions import EnumDefinition, EnumMemberDefinition
>>> status = EnumDefinition(
...     label_enum_typename="StatusLabel",
...     enhanced_enum_typename="EnhancedStatus",
...     value_type_typename="std::string_view",
...     members=[
...         EnumMemberDefinition(
...             enumerator_name="INITIALIZING",
...             enumerator_value_constant_name="INITIALIZING_VALUE",
...             enumerator_value_initializers="initializing",
...         ),
...         EnumMemberDefinition(
...             enumerator_name="WAITING_FOR_INPUT",
...             enumerator_value_constant_name="WAITING_FOR_INPUT_VALUE",
...             enumerator_value_initializers="waitingForInput",
...         ),
...         EnumMemberDefinition(
...             enumerator_name="BUSY",
...             enumerator_value_constant_name="BUSY_VALUE",
...             enumerator_value_initializers="busy",
...         ),
...     ],
...     associate_namespace_name="Statuses",
... )
>>> import enumecg
>>> enumecg.generate(status)
'...'
```

Note that in this style all names used in the C++ template are explicit fields of the `EnumDefinition` object.

3.2.2 Enum definitions in detail

Various aspects of code generation can be controlled by passing keyword arguments to the code generator functions.

Please note that when generating the code directly from *enumecg.definitions.EnumDefinition* object, the options have no effect because the `EnumDefinition` object is assumed to contain all information required to generate the code already.

Identifiers

In the example above the type names follow CamelCase while the enumerator names are UPPER_SNAKE_CASE. The code generator tries to deduce the case style for the different kinds of identifiers and uses it to format the names of others.

- The case style of the enum type name is used to format the names of the C++ enums and the associate namespace.
- The case style of the enumerator names are used to format the names of the the C++ enumerators and value constants.

The following case styles are recognized:

- Snake case with all lowercase letters: `lower_snake_case`
- Snake case with all uppercase letters: `UPPER_SNAKE_CASE`
- Camel case with every word capitalized: `CamelCase`
- Camel case with the first word starting with a lowercase letter: `mixedCase`. A single lower case word is recognized as snake_case instead of mixedCase.

Only ASCII alphanumeric characters are supported. Numbers may appear in any other position except at the start of a subword. All enumerators must follow the same case style. The following leads to an error:

```
>>> class BadExample(enum.Enum):
...     mixedCaseValue = "value1"
...     snake_case_value = "value2"
>>> enumecg.generate(BadExample)
Traceback (most recent call last):
  ...
enumecg.exceptions.Error: Could not find common case
```

Primary enum type

By default the label enum for `Status` has the name `StatusLabel` and the enhanced enum has the name `EnhancedStatus`. Almost certainly the user will want to call one of those types simply `Status` depending on the view whether the label enum or the enhanced enum is considered the *primary enum type*.

To make the label enum the primary type, set `primary_type` option to "label" when invoking the code generation:

```
>>> enumecg.generate(Status, primary_type="label")
'...enum class Status {...'
```

Similarly, passing option "enhanced" will make the enhanced enum the primary type:

```
>>> enumecg.generate(Status, primary_type="enhanced")
'...struct Status : ::enhanced_enum::enum_base<...'
```

Enumerator types and values

Python has dynamic typing, but in C++ all enumerators within an enum type must have the same type known in advance. There are two ways to define the enumerator type:

- Have the code generator deduce a C++ type automatically from the Python enumerator values
- Specify it manually

Enumerator type deduction

In the examples above the enumerator values are strings, but the enumerator type can be any type that can be constexpr constructible from arbitrarily nested initializer lists of string, integer, float and bool literals.

For example:

```
>>> class MathConstants(enum.Enum):
...     PI = 3.14
...     NEPER = 2.71
>>> enumecg.generate(MathConstants)
'...enum_base<..., double>...'
```

Or even:

```
>>> class NestedExample(enum.Enum):
...     EXPLICIT_VALUE = 0, ("string", True)
...     DEFAULT_VALUE = ()
>>> enumecg.generate(NestedExample)
'...enum_base<..., std::tuple<long, std::tuple<std::string_view, bool>>>...'
```

The Python types are mapped to C++ types in the following way:

- Integral types are mapped to `long`
- Other real numbers (like floats) are mapped to `double`
- `str` and `bytes` are mapped to `std::string_view`
- `bool` is mapped to `bool`
- Sequences are mapped to `std::tuple` whose template arguments are (recursively) the mapped types of the elements of the sequence.

All enumerator values must have a compatible types for the type deduction to work. When deducing the type from multiple sequences, the longest sequence determines the template arguments of the resulting `std::tuple`, and all prefixes of values must have types compatible with the longest sequence. For example the following works:

```
>>> class GoodExample(enum.Enum):
...     VALUE1 = 1, 2
...     VALUE2 = 3,
>>> enumecg.generate(GoodExample)
'...enum_base<..., std::tuple<long, long>>...'
```

But the following doesn't:

```
>>> class BadExample(enum.Enum):
...     VALUE1 = 1, 2
```

(continues on next page)

(continued from previous page)

```
...     VALUE2 = "string",
>>> enumecg.generate(BadExample)
Traceback (most recent call last):
  ...
enumecg.exceptions.Error: Could not deduce compatible type
```

Specifying enumerator type manually

You can use an type as the enum value type. Simply pass `value_type` option when invoking the code generation:

```
>>> enumecg.generate(Status, value_type="StatusValue")
'...enum_base<..., StatusValue>...'
```

`StatusValue` must be constexpr constructible from the `Status` member values, i.e. string literals. Let's look into that closer in the next section.

Enumerator value initialization

Warning: Converting Python object representations into C++ literals is done in a very straight-forward manner from the built-in `repr()`. It may not handle edge cases correctly, leading to compilation errors.

C++ enumerators are initialized with expressions based on the enum members used as arguments to the generator.

```
>>> enumecg.generate(Status)
'...value_type { "initializing" }...'
```

Sequences are converted to initializer lists recursively. Empty sequences are simply converted to initializer lists. Using the `NestedExample` above:

```
>>> enumecg.generate(NestedExample)
'...       value_type { 0, { "string", true } },\n        value_type {  },\n...'
```

Note that when generating the initializers, the underlying type is no longer considered. The generator just examines the values and converts them to possibly nested lists surrounded by braces. Thus empty tuple assigned to `NestedExample.DEFAULT_VALUE` was converted to an empty initializer list, i.e. the corresponding C++ enumerator is value initialized.

Overriding arbitrary fields in the definition

It is also possible to start with an *enumecg.definitions.EnumDefinition* object generated from any of the above representations, and modifying it before actually using it to generate the C++ code. *enumecg.definitions.make_definition()* can first be used to get an `EnumDefinition` object, which can further be used with the *enumecg.generate()* function.

Including documentation in the generator output

Doxygen comments can be included by using the `documentation` option:

```
>>> enumecg.generate(Status, documentation="doxygen")
'/** \\brief ...'
```

The generated documentation contains information about the usage of an enhanced enum type. The doxygen documentation of *Primary enum type* also includes the possible docstring of the Python enum.

3.2.3 Command line interface

The *enumecg* module can be invoked as a command. Given a YAML file `status.yaml`:

```
typename: Status
members:
- name: INITIALIZING
  value: initializing
- name: WAITING_FOR_INPUT
  value: waitingForInput
- name: BUSY
  value: busy
```

The command line interface can use this file as an input to print the generated code to stdout.

```
$ enumecg status.yaml
... # C++ boilerplate printed to stdout
```

The input file is a single YAML document containing an enum definition. See *Creating C++ enum from a mapping* for the details of the schema.

Invoking `enumecg --help` will list the supported options and arguments.

3.2.4 High level API

Most of the time the high level API is all you need to get started with code generation.

Generate Enhanced Enum definitions for C++

The top level module provides the high level code generation API for the Enhanced Enum library.

enumecg.**generate**(*enum: Union[*enumecg.definitions.EnumDefinition*, Mapping, enum.EnumMeta]*, ***, *documentation: Optional[Union[*enumecg.generators.DocumentationStyle*, str]] = None*, *primary_type: Optional[Union[*enumecg.definitions.PrimaryType*, str]] = None*, *value_type: Optional[str] = None*) → str

Generate code for an enhanced enum

This function is a shorthand for creating and invoking a code generator in one call.

The enum definition may be:

- An instance of *definitions.EnumDefinition*
- A `dict` object containing the enum definition. The required and optional keys are discussed in *Creating C++ enum from a mapping*.

- A native Python `enum.Enum` class. The typename is derived from the name of the enum class, and the enumerator definitions are derived from its members.

The exact way that the `enum` parameter is converted to an enum definition in the C++ code is covered in *Enum definitions in detail*.

Parameters

- **enum** – The enum definition
- **documentation** – A string or an enumerator indicating the documentation style. See *Including documentation in the generator output*.
- **primary_type** – A string or an enumerator indicating the primary type. See *Primary enum type*.
- **value_type** – See *Specifying enumerator type manually*.

Returns The enhanced enum definition created from the `enum` description.

enumecg.**generator**(*, *documentation: Optional[Union[*enumecg.generators.DocumentationStyle*, str]] = None*) → *enumecg.generators.CodeGenerator*

Create code generator for an enhanced enum type

Creates an instance of *generators.CodeGenerator*.

Parameters **documentation** – A string or an enumerator indicating the documentation style. See *Including documentation in the generator output*.

Returns The *generators.CodeGenerator* instance.

3.2.5 Module reference

The package contains lower level modules. These are used to implement the *High level API*, but can also be utilized directly to give greater control over the generated code.

Enum definitions

Contains the classes that the code generator uses as its representation of an enum definition.

enumecg.definitions.**Enum**

Generic enum definition

Types accepted by *make_definition()* and other functions that are used to generate enhanced enum definition.

alias of `Union[`*enumecg.definitions.EnumDefinition*, `Mapping, enum.EnumMeta]`

class enumecg.definitions.**EnumDefinition**(*label_enum_typename: str*, *enhanced_enum_typename: str*, *value_type_typename: str*, *members: Sequence[*enumecg.definitions.EnumMemberDefinition*]*, *associate_namespace_name: str*, *label_enum_documentation: Optional[*enumecg.definitions.EnumDocumentation*] = None*, *enhanced_enum_documentation: Optional[*enumecg.definitions.EnumDocumentation*] = None*)

Enum definition

class enumecg.definitions.**EnumDocumentation**(*short_description: Optional[str]*, *long_description: Optional[str]*)

Documentation associated with an enum

class enumecg.definitions.**EnumMemberDefinition**(*enumerator_name: str*, *enumerator_value_constant_name: str*, *enumerator_value_initializers: Union[Sequence, str]*)

Enum member definition

class enumecg.definitions.**PrimaryType**(*value*)

Possible primary types when generating enum definitions

These are the accepted choices for the `primary_type` argument in *make_definition()*.

enhanced = 'enhanced'

Enhanced enum is the primary type

label = 'label'

Label enum is the primary type

enumecg.definitions.**make_definition**(*enum: Union[enumecg.definitions.EnumDefinition, Mapping, enum.EnumMeta]*, ***, *primary_type: Optional[enumecg.definitions.PrimaryType] = None*, *value_type: Optional[str] = None*) → *enumecg.definitions.EnumDefinition*

Make *EnumDefinition* instance from various types

This function is used to convert various kinds of enum definitions (standard Python `enum.Enum` types, `dict` instances etc.) into an *EnumDefinition* instance usable by the code generator. It allows for an user to provide a simpler enum definition, and having the details filled in automatically.

This function is mainly meant to be used by the high level functions in the top level *enumecg* module, but can also be invoked directly for greater control over the code generation process.

Parameters

- **enum** – The enum definition.
- **primary_type** – A *PrimaryType* enumerator indicating the primary type. See *Primary enum type*.
- **value_type** – See *Specifying enumerator type manually*.

Raises ***exceptions.Error*** – If `enum` is invalid and cannot be converted to *EnumDefinition*.

Code generator

The module contains the code generator consuming enum definitions and outputting C++ code.

class enumecg.generators.**CodeGenerator**(***, *documentation: Optional[enumecg.generators.DocumentationStyle] = None*)

Code generator for an enhanced enum type

Used to generate the necessary C++ boilerplate to make an enum type compatible with the Enhanced Enum library.

The recommended way to create an instance is by using the *enumecg.generator()* function.

Parameters **documentation** – A *DocumentationStyle* enumerator indicating the documentation style. See *Including documentation in the generator output*.

generate_enum_definitions(*enum*, ***options*)

Generate the C++ definitions needed for an enhanced enum

Parameters

- **enum** – The enum definition

- **options** – The options passed to `definitions.make_definition()`.

Returns The generated code

Raises *exceptions.Error* – If the code generation fails due to an invalid enum definition.

class enumecg.generators.**DocumentationStyle**(*value*)
Possible documentation styles

These are the accepted choices for the `documentation` argument in *CodeGenerator()*.

doxygen = 'doxygen'
Doxygen documentation style

Utilities

Utilities to perform miscellaneous tasks that the library needs to perform. While they are mainly targeted for internal use, they are may also be useful outside the scope of the *enumecg* package.

class enumecg.utils.**CppTypeDeducer**(**values*, *type_name: Optional[str] = None*)
Deduce C++ types and initializers from Python values

This class examines collections of Python values, and deduces a C++ type that is compatible with them. It implements the algorithm described in *Enumerator types and values*.

If the explicit `type_name` parameter is given, it is preferred and the `values` are not examined.

Parameters

- **values** – The values used to deduce the type
- **type_name** – The type name

Raises *exceptions.Error* – If no C++ type compatible with `values` can be deduced.

classmethod get_initializer(*value*)
Return C++ initializer for `value`

Parameters **value** – A value consisting of string, numbers, booleans and nested sequences thereof.

Returns An expression that can be used in a C++ initializer list to initialize a type compatible with `value` at compile time

property type_name: str
The deduced C++ type

class enumecg.utils.**NameFormatter**(**names: str*)
Format names in the same case style as sample names

This class is used to split a sample of names (variables, classes etc.) into subwords, and creating new names with the same case style. An example demonstrates this the best:

```
>>> formatter = NameFormatter("first_name", "second_name")
>>> formatter.parts
[['first', 'name'], ['second', 'name']]
>>> formatter.join(["name", "in", "snake", "case"])
'name_in_snake_case'
>>> formatter.join(["snake", "case"], pluralize=True)
'snake_cases'
```

This class implements the identifier formatting described in *Identifiers*.

Parameters **names** – The names to analyze

Raises ***exceptions.Error*** – If at least one of the `names` doesn't follow a known case style, or if the sample contains names that follow different case style.

join(*parts: Iterable[str]*, *, *pluralize=False*) → str
Create new name from `parts`

Parameters

- **parts** – Parts (words) of the name
- **pluralize** – If `True`, assume the argument is a singular noun, and return it pluralized.

Returns The new name as string, with the individual parts joined together using the case style inferred during the construction

property parts: List[List[str]]
List of the name parts used to create the formatter

Exceptions

Exceptions related to the code generation process.

exception enumecg.exceptions.**Error**
Generic error in the code generation process

CHAPTER

FOUR

DEVELOPER GUIDE

4.1 Building and installing from sources

The project uses CMake as its build system. To build everything:

```
$ cd /path/to/build
$ cmake                                     \
>   -D ENHANCEDENUM_BUILD_DOCS:BOOL=ON      \
>   -D ENHANCEDENUM_BUILD_PYTHON:BOOL=ON    \
>   -D ENHANCEDENUM_BUILD_TESTS:BOOL=ON     \
>   /path/to/repository
```

The Enhanced Enum library specific CMake variables are:

- `ENHANCEDENUM_BUILD_DOCS`: Build `sphinx` docs
- `ENHANCEDENUM_BUILD_PYTHON`: Build the *enumecg* package (see caveats below)
- `ENHANCEDENUM_BUILD_TESTS`: Build tests for the C++ and/or Python packages

The C++ headers under the `cxx/include/` directory will always be installed, along with CMake config files needed to find the package in other projects. When installed this way, the project is exposed as imported target `EnhancedEnum::EnhancedEnum`:

```
find_package(EnhancedEnum)
target_link_libraries(my-target EnhancedEnum::EnhancedEnum)
```

4.1.1 Python environment

Docs, EnumECG and unit tests *all* require Python when being built. This would be a typical way to bootstrap a virtual environment, and build and test the C++ code and documentations with CMake:

```
$ mkdir /path/to/build
$ cd /path/to/build
$ source /path/to/venv/bin/activate
$ pip install -r requirements.txt -r requirements-dev.txt
$ cmake                                     \
>   -D ENHANCEDENUM_BUILD_DOCS:BOOL=ON      \
>   -D ENHANCEDENUM_BUILD_TESTS:BOOL=ON     \
>   /path/to/repository
$ make && make test && make install
```

4.1.2 Installing EnumECG from sources

EnumECG uses Flit for building. By default CMake will not build the EnumECG library. If you want to install *enumecg* from sources using CMake, you can do that:

```
$ cd /path/to/build
$ cmake -D ENHANCEDENUM_BUILD_PYTHON:BOOL=ON /path/to/repository
$ make && make install
```

Installing the package in this way is limited. Essentially it's equivalent of running the following in the `python/` directory:

```
$ flit build
$ flit install
```

Note that the build will happen in-source.

CHAPTER

FIVE

HISTORY

5.1 Changelog

5.1.1 Version 0.8

Date 2022-02-17

Added

- Python 3.10 support for `enumecg`

Changed

- Make the range returned by `all()` implement the C++20 view concept

5.1.2 Version 0.7

Date 2021-06-22

Fixed

- `enumecg.utils.NameFormatter` now supports snake cased names that have numeric parts

5.1.3 Version 0.6

Date 2020-12-27

Added

- Three-way comparison operator for C++20 builds

Fixed

- Do not return dangling references from `enum_iterator::operator*()` and `enum_iterator::operator[]()`

5.1.4 Version 0.5

Date 2020-07-30

Added

- Pylint (test target, configurations)
- Add aliases of `begin()`, `end()` and `all()` to the associate namespace generated by EnumECG
- Add hash object for enhanced enums

Fixed

- Several pylint errors
- Fix bugs in the CMake build system

5.1.5 Version 0.4

Date 2020-05-13

Added

- Command line interface

Changed

- Migrate EnumECG build system to flit
- Migrate EnumECG unit tests to pytest
- Use tox to manage EnumECG unit tests
- More explicit call signatures in Python API (more type annotations, less catch-all keyword arguments)
- Change the format of `dict` representation of enumerators to have a more explicit ordering of members
- Improvements to the documentation
- Use Python enums to enumerate the possible primary types and documentation styles in the EnumECG library

5.1.6 Version 0.3

Date 2020-03-15

Added

- Docstring from the Python enum definition is now included in the generated Doxygen comments
- Documentation about integrating EnumECG to a project

Changed

- Restructured documentation slightly

5.1.7 Version 0.2

Date 2020-01-10

Added

- Add *enhanced_enum::enum_base::ssize()*
- Add *enhanced_enum::enum_base::begin()* and *enhanced_enum::enum_base::end()*
- Add support for generating Doxygen comments

Changed

- Implement *enhanced_enum::enum_base::all()* in terms of custom range type (not array)

Fixed

- Add include guards to the C++ headers

5.1.8 Version 0.1

Date 2019-12-07

Initial revision

PYTHON MODULE INDEX

e

INDEX

M

N

P

T