Construction Rules for enum class Values

Gabriel Dos Reis
Microsoft

Abstract

This paper suggests a simple adjustment to the existing rules governing conversion from the underlying type of a scoped enumeration to said enumeration, if the latter is defined with no associated enumerator. This effectively supports programming styles that rely on defining of new distinct integral types based out of existing integer types, without the complexity of anarchic integer conversions, while retaining all the ABI characteristics and benefits of the integer types, especially for system programming.

1 INTRODUCTION

There is an incredibly useful technique for introducing a new integer type that is almost an exact copy, yet distinct type in modern C++11 programs: an `enum class` with an explicitly specified underlying type. Example:

```
enum class Index : uint32_t { };    // Note: no enumerator.
```

One can use `Index` as a new distinct integer type, it has no implicit conversion to anything (good!) This technique is especially useful when one wants to avoid the anarchic implicit conversions C++ inherited from C. For all practical purposes, `Index` acts like a "strong typedef" in C++11.

There is however, one inconvenience: to construct a value of type `Index`, the current language spec generally requires the use a cast -- either a `static_cast` or a functional notation cast. This is both conceptually wrong and practically a serious impediment. Constructing an `Index` value out of `uint32_t` is not a cast, no more than we consider

```
struct ClassIndex { uint32_t val; };

ClassIndex idx { 42 };
```

a cast. It is a simple construction of a value of type `ClassIndex`, with no narrowing conversion. I claim the current rule for scoped enumeration is too strict. For instance, we should be able to write

```
int f(Index);
auto a = f({42});
```

This proposal suggests we allow an implicit/non-narrowing conversion from a scoped enumeration's underlying type to the enumeration itself, when its definition introduces no enumerator and the source uses a list-initialization syntax. This is safe and support very useful programming techniques. For example, you could introduce new integer types (e.g. `SafeInt`) that ***enjoy the same existing calling conventions as its underlying integer type***, even on ABIs expressly designed to penalize passing/returning structures by value. This supports a zero-overhead abstraction technique. It has been found very popular in practice by system programmers and application programmers.

Strictly speaking, this change could be detected by SFINAE tricks; however, the benefit is much greater -- and the SFINAE trick detection is more useful in the *other* direction, which I am not proposing to change.

2 Changes from Previous Versions

This paper was reviewed by EWG at the Fall 2015 meeting in Kona, HI. It was approved by EWG for C++17. The original wording was reviewed by CWG at that meeting, with suggested tweaks and design questions for EWG. The design questions were resolved by EWG.

- The phrase "integer class" was removed. That phrase was introduced to designate existing construct in the language (scoped enums with no enumerators), but it appears to cause confusion as to whether the proposal was suggesting a new type definition mechanism. It was and is not.
- The elision of the type name is no longer permitted in function calls, for backward compatibility.
- The construct is allowed also for traditional enums with fixed underlying type.

EWG did consider the request of extending the relaxation suggested in this paper to enumerations with declared enumerators, but ultimately rejected that suggestion.

3 Wording

Add a bullet between (3.8) and (3.9) to paragraph 8.5.4/3 as follows:

> Otherwise, if T is an enumeration with a fixed underlying type (7.2) with no declared enumerators, the initializer list shall be either empty or of the form { v } and the conversion from v to E (if any) shall not involve a narrowing conversion. In either case, the object is initialized with T() if the initializer list was empty, or the functional cast expression T(v). This allowance is limited to the contexts of initializer in a variable definition, initializer in a *new-expression*, initializer of a non-static data member, *mem-initializer*, a *brace-init-list* following an enumeration type name (5.2.3). [Example:
>
> enum byte : unsigned char { };
>
> byte b { 42 };   // OK
>
> byte c = { 42 };   // OK; same value as b
>
> byte d = byte{ 42 };   // OK; same value as b

```
void f(byte);

f({ 42 });                                   // error

--end example]
```

4 ACKNOWLEDGMENT

This proposal formalizes the TINY suggestion made on EWG reflector [1]. It benefited from feedback from various people, in particular Richard Smith and Jens Maurer. After the draft of this paper was completed, I was made aware of the paper authored by Walter Brown reviving the suggestion of "opaque typedef" [2]. The current suggestion is not incompatible with Walter's proposal, nor is it a replacement or a competing proposal. It does not provide any new way of introducing types or type names. The only novelty (a valuable one!) is the removal of some rules surrounding value construction of certain enums. This proposal is more of a completion of Oleg Smolsky's proposal [3], but for enumerations.

5 REFERENCES

[1] Gabriel Dos Reis, *[TINY] enum class conversion and conversion from underlying type when no enumerator is introduced,* 2015. Reflector message c++std-ext-16296 posted on January 7, 2015.

[2] Walter Brown, "Function Aliases + Extended Inheritance = Opaque Typedefs," ISO/IEC JTC1/SC22/WG21, 2015. Document number P0109R0

[3] Oleg Smolsky, "Extension to aggregate initialization," ISO/IEC JTC1/SC22/WG21, 2015. Document numnber P0017R0