Adding enum struct: Name-Spaced Enumerations for C

Author: Yair Lenga

Email: yair.lenga@gmail.com

Date: 2025-06-08

Document Number: N3568

Project: Programming Language C

Proposal Category: Language extension

Target Audience: WG14

Abstract

This proposal introduces a new declaration form `enum struct` that groups integer constants in a scoped namespace. It enables symbolic constants without polluting the global namespace and is modeled after C++'s `enum class`. This feature improves code safety and clarity in large C projects.

Proposal

We propose adding `enum struct` to C as a scoped enumeration structure. This construct defines a set of named integer constants within a namespace and acts as a compile-time constant object. It is similar in usage to `constexpr struct` in C23, but designed to support grouped symbolic values with minimal syntax.

This feature is conceptually consistent with the `constexpr` feature introduced in C23. `enum struct` can be viewed as syntactic sugar for defining a compile-time constant object with a sequence of integer values. It offers a concise and semantically scoped way to express a list of symbolic constants, avoiding manual use of `constexpr int` declarations or `#define` macros.

Syntax

`enum struct` follows this syntax:

```
enum struct Name {
    IDENTIFIER [= VALUE], ...
};
```

Values default to 0-based increment unless specified. Explicit assignments allow skipping or custom ordering.

Semantics

`enum struct` defines both an enum type (`enum Name`) and a constexpr-like object (`Name`) holding the constants.

Each member is assigned an integer value, starting from 0 and incremented by 1 unless explicitly overridden.

This provides strongly-typed symbolic constants, with syntax clarity and scoped naming.

Model and Interpretation

`enum struct` behaves like a constexpr struct of `int` values. Each identifier becomes a compile-time constant, usable in constant expressions, switch statements, or array sizes.

```
enum struct Status { OK = 0, FAIL, TIMEOUT };
  int x = Status.FAIL; // OK
```

No new runtime semantics are introduced — it is a purely syntactic addition that builds on existing enum rules.

Motivation: Real-World Constants

In large C codebases and system headers, it is common to encounter extensive sets of constants defined using `#define` macros, often with prefix-based naming conventions to avoid name collisions. These flat constant lists lead to global namespace pollution, inconsistent scoping, and difficulty in managing or grouping related values.

POSIX errno constants:
#define EPERM 1
#define ENOENT 2
...

With `enum struct`:

```
  enum struct Errno { PERM = 1, NOENT = 2 };
```

Similar benefit applies to HTTP status codes, SQLite result codes, and Windows error definitions.

In short – the proposal provide a path for easier integration of large number of libraries – reducing the risk that #define name collision, and allowing each for cleaner code – as it will reduce the need to “prefix” everything.

Usage and Portability

`enum struct` can be used in both global and local scope. Constants may be aliased using `constexpr int LOCAL = Name.VALUE;` to bring values into scope explicitly.

```
enum struct Status { OK, FAIL };
  constexpr int OK_LOCAL = Status.OK;
```

This usage aligns with `constexpr` and supports portability to environments with or without full `enum struct` support.

Relationship to C23 constexpr

Conceptually, the construct 'enum struct'

```
enum struct FOO {
   E1=3,
   E2,
   E3
} ;
```

Will be equivalent to the following, where TTT is a unique ID.

```
enum FOO {
    FOO_TTT_E1=3,
    FOO_TTT_E2,
    FOO_TTT_E3
} ;

struct FOO_TTT {
    enum FOO E1;
    enum FOO E2;
    enum FOO E3;
} ;

constexpr struct FOO_TTT FOO = {
    .E1 = FOO_TTT_E1,
    .E2 = FOO_TTT_E2,
    .E3 = FOO_TTT_E3
};
```

Backward Compatibility

The proposal introduces no changes to existing `enum` or macro behavior. Libraries can adopt `enum struct` incrementally. Projects with controlled build environments are likely to benefit from early adoption.

The proposal does NOT create new namespaces in the C language – it's using the existing one: ordinary identifier (variables/functions/typedefs), tags (struct/union/enum), struct members, labels and macro).

When 'struct enum FOO' will be used, FOO entry will be added to the tags list and ordinary variable list. It will be an error to try to create another structure or enum or union with the same name. This is compatible with the current C23 rules which do NOT allow using the same tag for struct, enum or union.

C++ Compatibility

This feature supports migration toward cleaner C/C++ interoperability and symbolic hygiene.

- The declaration enum struct …' is VALID C++.
- The "expanded" enum/struct/constexpr construct is also a valid "C++".
- The proposal does NOT allow the similar, 'num class',
- In C++, the member selection operator is '::' (e.g., FOO::E1). In C, we will continue to use '.' (FOO.E1).

Rationale

C lacks a mechanism for grouping symbolic constants without polluting the global namespace. As a result, common C headers and libraries rely on prefix-heavy macros (e.g., `HTTP_OK`, `SQLITE_FAIL`, `ERROR_BUSY`) or unscoped `enum` types with naming conventions. These patterns lead to fragile code, naming conflicts, and unnecessary verbosity, particularly in large or modular codebases.

The proposed `enum struct` construct introduces a clean, scoped namespace for compile-time constants. It mirrors the structure and usage of C++'s `enum class` while remaining within the constraints of the C type and value model. The semantics are consistent with C23's `constexpr`—in fact, `enum struct` is syntactic sugar for defining a compile-time constant object with auto-incrementing integer fields.

This feature enables a header-safe, symbol-safe mechanism for defining logically grouped constants, without sacrificing readability, performance, or tool compatibility. Existing enums and macros remain unaffected. The addition is purely additive and backward-compatible.

Other options such as `#define`, long name prefixes, or `typedef struct` patterns lack the namespace safety and compile-time clarity this proposal provides. `enum struct` represents a small but meaningful step toward cleaner, safer code in modern C.