The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.7.2.2 Enumeration specifiers

enumera-
tion specifier
syntax


```
enum-specifier:
                enum identifier_opt { enumerator-list }
                enum identifier_opt { enumerator-list , }
                enum identifier
enumerator-list:
                enumerator
                enumerator-list , enumerator
enumerator:
                enumeration-constant
                enumeration-constant = constant-expression
```

Commentary

Support for a trailing comma is intended to simplify the job of automatically generating C source.

C90

Support for a trailing comma at the end of an *enumerator-list* is new in C99.

C++

The form that omits the brace enclosed list of members is known as an elaborated type specifier, 7.1.5.3, in C++.

The C++ syntax, 7.2p1, does not permit a trailing comma.

Other Languages

Many languages do not use a keyword to denote an enumerated type, the type is implicit in the general declaration syntax. Those languages that support enumeration constants do not always allow an explicit value to be given to an enumeration constant. The value is specified by the language specification (invariably using the same algorithm as C, when no explicit values are provided).

Common Implementations

Support for enumeration constants was not included in the original K&R specification (support for this functionality was added during the early evolution of C[6]). Many existing C90 implementations support a trailing comma at the end of an *enumerator-list*.

Coding Guidelines

enumeration
set of named
constants

A general discussion on enumeration types is given elsewhere.

The order in which enumeration constants are listed in an enumeration type declaration often follows some rule, for instance:

- *Application conventions* (e.g., colors of rainbow, kings of England, etc.).
- *Human conventions* (e.g., increasing size, direction— such as left-to-right, or clockwise, alphabetic order, etc.).
- *Numeric values* (e.g., baud rate, Roman numerals, numeric value of enumeration constant, etc.).

developer
expectations

While ordering the enumeration constant definitions according to some rule may have advantages (directly mapping to a reader's existing knowledge or ordering expectations may reduce the effort needed for them to organize information for later recall), there may be more than one possible ordering, or it may not be possible to create a meaningful ordering. For this reason no guideline recommendation is made here.

init-declarator ??
one per source line

Do the visual layout factors that apply to the declaration of objects also apply to enumeration constants? The following are some of the differences between the declarations of enumeration constants and objects:

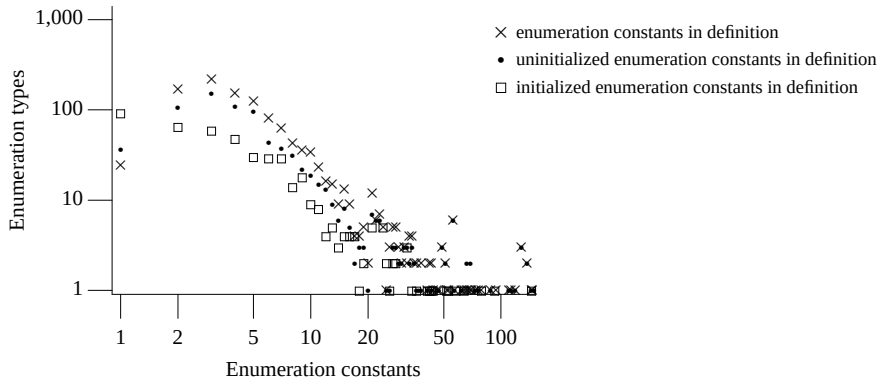

Figure 1439.1: Number of enumeration constants in an enumeration type and number whose value is explicitly or implicitly specified. Based on the translated form of this book's benchmark programs (also see Figure **??**).

- There are generally significantly fewer declarations of enumerator constants than objects, in a program (which might rule out a guideline recommendation on the grounds of applying to a construct that rarely occurs in source).
- Enumeration constants are usually declared amongst other declarations at file scope (i.e., they are not visually close to statements). One consequence of this is that, based on declarations being read on as as-needed basis, the benefits of maximizing the amount of surrounding code that appears on the display at the same time are likely to be small.

reading kinds of

The following guideline recommendation is given for consistency with other layout recommendations.

Cg 1439.1 No more than one enumeration constant definition shall occur on each visible source code line.

The issue of enumeration constant naming conventions is discussed elsewhere.

enumeration constant naming conventions

Usage

A study by Neamtiu, Foster, and Hicks[4] of the release history of a number of large C programs, over 3-4 years (and a total of 43 updated releases), found that in 40% of releases one or more enumeration constants were added to an existing enumeration type while enumeration constants were deleted in 5% of releases and had one or more of their names changed in 16% of releases.[3]

Table 1439.1: Some properties of the set of values (the phrase *all values* refers to all the values in a particular enumeration definition) assigned to the enumeration constants in enumeration definitions. Based on the translated form of this book's benchmark programs.

Property	%
All value assigned implicitly	60.1
All values are bitwise distinct and zero is not used	8.6
One or more constants share the same value	2.9
All values are continuous , i.e. , number of enumeration constants equals maximum value minus minimum value plus 1	80.4

Constraints

1440 The expression that defines the value of an enumeration constant shall be an integer constant expression that has a value representable as an **`int`**.

enumeration constant representable in int

Commentary

constant representable in its type
enumeration constant type

This constraint is consistent with the requirement that the value of a constant be in the range of representable values for its type. Enumeration constants are defined to have type **int**.

C++

7.2p1 *The* `constant-expression` *shall be of integral or enumeration type.*

7.2p4 *If an initializer is specified for an enumerator, the initializing value has the same type as the expression.*

Source developed using a C++ translator may contain enumeration initialization values that would be a constraint violation if processed by a C translator.

```
#include <limits.h>

enum { umax_int = UINT_MAX}; /* constraint violation */
                             // has type unsigned int
```

Common Implementations

Some implementations support enumeration constants having values that are only representable in the types **unsigned int**, **long**, or **unsigned long**.

Coding Guidelines

The requirement is that the constant expression have a value that is representable as an **int**. The only requirement on its type is that it be an integer type. The constant expression may have a type other than **int** because of the use of a macro name that happens to have some other type, or because one of its operands happens to have a different type. If the constant expression consists, in the visible source, of an integer constant containing a suffix, it is possible that the original author or subsequent readers may assume some additional semantics are implied. However, such occurrences are rare and for this reason no guideline covering this case is given here.

There may be relationships between different enumeration constants in the same enumeration type. The issue of explicitly showing this relationship in the definition, using the names of those constants rather than purely numeric values, is a software engineering one and is not discussed further in these coding guidelines.

```
enum { E1 = 33, E2 = 36, E3 = 3 };

/* does not specify any relationship, and is not as resistant to modification as: */

enum { e1 = 33, e2 = e1+3, e3 = e2-e1 };
```

The enumeration constants defined in by an enumerated type are a set of identifiers that provide a method of naming members having a particular property. These properties are usually distinct and in many cases the values used to represent them are irrelevant.

Semantics

enumerators type int

The identifiers in an enumerator list are declared as constants that have type **int** and may appear wherever such are permitted.[107)] 1441

Commentary

enumeration constant type
expression wherever an int may be used

The issues associated with enumeration constants having type **int** are discussed elsewhere, as are the issues of it appearing wherever such a type is permitted.

C++

Following the closing brace of an `enum-specifier`*, each enumerator has the type of its enumeration. Prior to the closing brace, the type of each enumerator is the type of its initializing value.* 7.2p4

In C the type of an enumeration constant is always **int**, independently of the integer type that is compatible with its enumeration type.

```
#include <limits.h>

int might_be_cpp_translator(void)
{
enum { a = -1, b = UINT_MAX }; // each enumerator fits in int or unsigned int

return (sizeof(a) != sizeof(int));
}

void CPP_DR_172_OPEN(void) // Open C++ DR
{
enum { zero };

if (-1 < zero) /* always true */
                // might be false (because zero has an unsigned type)
   ;
}
```

Other Languages

Most languages that contain enumerator types treat the associated enumerated constants as belonging to a unique type that is not compatible with type **int**. In these languages an enumeration constant must be explicitly cast (Pascal provides a built-in function, **ord**) before they can appear where a constant having type **int** may appear.

Coding Guidelines

The values given to the enumeration constants in a particular enumeration type determine their role and the role of an object declared to have that type. To fulfil a bit-set role the values of the enumeration constants need to be bitwise distinct. All other cases create a type that has a symbolic role.

object role

bit-set role

Example

```
enum T { attr_a = 0x01, attr_b = 0x02, attr_c = 0x04, attr_d = 0x10, attr_e = 0x20};
```

1442 An enumerator with = defines its enumeration constant as the value of the constant expression.

Commentary

This specifies the semantics associated with a token sequence permitted by the syntax (like the semantics of simple assignment, the identifier on the left of the = has as its value the constant expression on the right).

Other Languages

Not all languages that support enumeration constants allow the value, used to represent them during program execution, to be specified in their definition.

Coding Guidelines

Some guideline documents recommend against assigning an explicit value to an enumeration constant. Such recommendations limit enumeration types to having a symbolic role only. It has the effect of giving developers no choice but to use object-like macros to create sets of identifiers having bit-set roles. Using macros instead of enumerations makes it much more difficult for static analysis tools to deduce an association between identifiers (it may still be made apparent to human readers by grouping of macro definitions and appropriate commenting), which in turn will reduce their ability to flag suspicious use of such identifiers.

macro object-like

If the first enumerator has no =, the value of its enumeration constant is 0. 1443

Commentary

This choice is motivated by common usage and implementation details. Most enumeration types contain relatively few enumeration constants and many do not explicitly assign a value to any of them. Implicitly starting at zero means that it is possible to represent the values of all enumeration constants in an enumeration type within a byte of storage.

limit enumeration constants

Other Languages

This is the common convention specified by other languages, or by implementations of other languages that do not specify the initial value.

Each subsequent enumerator with no = defines its enumeration constant as the value of the constant expression obtained by adding 1 to the value of the previous enumeration constant. 1444

Commentary

If the previous enumeration constant had the value MAX_INT, adding one will produce a value that cannot be represented in an **int**, violating a constraint.

enumeration constant representable in int 1440

Other Languages

This is the common convention specified by other languages, or by implementations of other languages that do not specify the initial value.

(The use of enumerators with = may produce enumeration constants with values that duplicate other values in the same enumeration.) 1445

Commentary

When such enumeration constants are tested for equality with each other the result will be 1 (true), because it is their values not their spellings that are compared.

C++

The C++ Standard does not explicitly mention this possibility, although it does give an example, 7.2p2, of an enumeration type containing more than one enumeration constant having the same value.

Other Languages

No languages known to your author, that support the explicit definition of enumeration constant values, prohibits the appearance of duplicate values in the same enumeration.

Coding Guidelines

There are two ways in which more than one enumeration constant, in the same enumerated type, can have the same value. Either the values were explicitly assigned, or the at least one of the values was implicitly assigned its value. This usage may be an oversight, or it may be intentional (i.e., fixing the names of the first and last enumeration constant when it is known that new members may be added at a later date). These guideline recommendations are not intended to recommend against the creation of faults in code. What of the intended usage?

guidelines not faults

```
1   enum ET {FIRST_YEAR, Y_1898=FIRST_YEAR, Y_1899, Y_1900, LAST_KNOWN_YEAR=Y1900};
```

Do readers of the source assume there are no duplicate values among different enumeration constants, from the same enumerated type? Unfortunately use of enumerations constants are not sufficiently common among developers to provide the experience needed to answer this question.

1446 The enumerators of an enumeration are also known as its members.

Commentary

Developers often refer to the enumerators as *enumeration constants*, rather than *members*.

C++

The C++ Standard does not define this additional terminology for enumerators; probably because it is strongly associated with a different meaning for members of a class.

. . . the associated `enumerator` *the value indicated by the* `constant-expression`. 7.2p1

1447 Each enumerated type shall be compatible with **`char`**, a signed integer type, or an unsigned integer type.

enumeration type compatible with

Commentary

This is a requirement on the implementation. The term *integer types* cannot be used because enumerated types are included in its definition. There is no guarantee that when the **`sizeof`** operator is applied to an enumerator the value will equal that returned when **`sizeof`** is applied to an object declared to have the corresponding enumerator type.

integer types

C90

Each enumerated type shall be compatible with an integer type;

The integer types include the enumeration types. The change of wording in the C99 Standard removes a circularity in the specification.

integer types

C++

An enumeration is a distinct type (3.9.1) with named constants. 7.2p1

The underlying type of an enumeration may be an integral type that can represent all the enumerator values defined in the enumeration (7.2p5). But from the point of view of type compatibility it is a distinct type.

enumeration constant type
enumeration different type

It is implementation-defined which integral type is used as the underlying type for an enumeration except that the underlying type shall not be larger than ***`int`*** *unless the value of an enumerator cannot fit in an* ***`int`*** *or* ***`unsigned int`***. 7.2p5

While it is possible that source developed using a C++ translator may select a different integer type than a particular C translator, there is no effective difference in behavior because different C translators may also select different types.

Other Languages

Most languages that support enumerated types treat such types as being unique types, that is not compatible with any other type.

Coding Guidelines

Experience shows that developers are often surprised by some behaviors that occur when a translator selects a type other than **int** for the compatible type. The two attributes that developers appear to assume an enumerated type to have are promoting to a signed type (rather than unsigned) and being able to represent all the values that type **int** can (if values other than those in the enumeration definition are assigned to the object).

If the following guideline recommendation on enumerated types being treated as not being compatible with any integer type is followed, these assumptions are harmless.

Experience with enumerated types in more strongly typed languages has shown that the diagnostics issued when objects having these types, or their members, are mismatched in operations with other types, are a very effective method locating faults. Also a number of static analysis tools[1,2,5] perform checks on the use of objects having an enumerated type and their associated enumeration constants[1447.1] .

Cg 1447.1 Objects having an enumerated type shall not be treated as being compatible with any integer type.

Example

```
1  #include <stdio.h>
2
3  void f(void)
4  {
5  enum T {X};
6
7  if ((enum T)-1 < 0)
8     printf("The type of enum {X} is signed\n");
9
10 if (sizeof(enum T) == sizeof(X))
11    printf("The type of enum {X} occupies the same number of bytes as int\n");
12 }
```

The choice of type is implementation-defined,[108)] but shall be capable of representing the values of all the members of the enumeration. 1448

Commentary

This is a requirement on the implementation.

C90

The requirement that the type be capable of representing the values of all the members of the enumeration was added by the response to DR #071.

Other Languages

Languages that support enumeration types do not usually specify low level implementation details, such as the underlying representation.

Common Implementations

Most implementations chose the type **int**. A few implementations attempt to minimize the amount of storage occupied by each enumerated type. They do this by selecting the compatible type to be the integer type with the lowest rank, that can represent all constant values used in the definition of the contained enumeration constants.

[1447.1] However, this is not necessarily evidence of a worthwhile benefit. Vendors do sometimes add features to a product because of a perceived rather actual benefit.

Coding Guidelines

An implementation's choice of type will affect the amount of storage allocated to objects defined to have the enumerated type. The alignment of structure members having such types may also be affected. One reason why some developers do not use enumerated types is that they do not always have control over the amount of storage allocated. While this is a very minor consideration in most environments, in resource constrained environments it may be of greater importance.

A definition of an enumeration type may not include (most don't) enumeration constants for each of the possible values that can be represented in the underlying value representation (invariably some integer type). The guideline recommendation that both operands of a binary operator have the same enumerated type limits, but does not prevent, the possibility that a value not represented in the list of enumeration constants will be created.

?? enumeration constant as operand

Whether or not the creation of a value that is not represented in the list of enumeration constants is considered to be acceptable depends on the interpretation given to what *value* means. The approach taken by these coding guideline subsections is to address the issue from the point of view of the operators that might be expected to apply to the given enumeration type (these are discussed in the C sentence for the respective operators). The following example shows two possibilities:

```
1  enum roman_rep {
2                   I = 1,
3                   V = 5,
4                   X = V+V,
5                   L = V*X,
6                   C = L+L,
7                   D = C*V,
8                   M = X*C
9                  } x;
10 enum termios_c_iflag {              /* A list of bitwise distinct values. */
11                       BRKINT = 0x01,
12                       ICRNL  = 0x02,
13                       IGNBRK = 0x04,
14                       IGNCR  = 0x10,
15                       IGNPAR = 0x20,
16                       INLCR  = 0x40
17                      } y;
18
19 void f(void)
20 {
21 x= V+V;             /* Create a value whose arithmetic value is     represented. */
22 x= X+L;             /* Create a value whose arithmetic value is not represented. */
23 y= ICRNL | IGNPAR;  /* Create a value whose bit-set is     represented. */
24 y= ICRNL << 8;      /* Create a value whose bit-set is not represented. */
25 }
```

1449 The enumerated type is incomplete until after the } that terminates the list of enumerator declarations.

enumerated type incomplete until

Commentary

This sentence is a special case of one given elsewhere.

tag incomplete until

C90

The C90 Standard did not specify when an enumerated type was completed.

C++

The C++ Standard neither specifies that the enumerated type is incomplete at any point or that it becomes complete at any point.

Following the closing brace of an `enum-specifier`*, each enumerator has the type of its enumeration*

Example

The definition:

```
enum e_tag { e1 = sizeof(enum e_tag)};
```

is not permitted (it is not possible to take the size of an incomplete type). But:

```
enum e_tag { e1, e2} e_obj[sizeof(enum e_tag)];
```

is conforming.

EXAMPLE The following fragment: 1450

```
enum hue { chartreuse, burgundy, claret=20, winedark };
enum hue col, *cp;
col = claret;
cp = & col;
if (*cp != burgundy)
        /* ... */
```

makes **hue** the tag of an enumeration, and then declares **col** as an object that has that type and **cp** as a pointer to an object that has that type. The enumerated values are in the set { 0, 1, 20, 21 }.

C++

The equivalent example in the C++ Standard uses the enumeration names red, yellow, green and blue.

Other Languages

In Pascal this example could be written as (no explicit assignment of values is supported):

```
type
     hue : ( chartreuse, burgundy, claret, winedark );
```

and in Ada as:

```
type
     hue is ( chartreuse, burgundy, claret, winedark );
for hue use (chartreuse => 0, burgundy => 1, claret => 20, winedark => 21);
```

Forward references: tags (6.7.2.3). 1451

---

footnote 107

107) Thus, the identifiers of enumeration constants declared in the same scope shall all be distinct from each other and from other identifiers declared in ordinary declarators. 1452

Commentary

name space ordinary identifiers
identifier no linkage
declaration only one if no linkage

This requirement can be deduced from the fact that enumeration constants are in the same name space as ordinary identifiers, they have no linkage, and that only one identifier with these attributes shall (a constraint) be declared in the same scope.

C++

The C++ Standard does not explicitly make this observation.

Other Languages

Ada permits the same identifier to be defined as an enumeration constant in different enumerated type in the same scope. References to such identifiers have to be explicitly disambiguated.

1453 108) An implementation may delay the choice of which integer type until all enumeration constants have been seen.

footnote 108

Commentary

This footnote is discussing the behavior of a translator that processes the source in a single pass. Many translators operate in a single pass and this behavior enables this form of implementation to continue to be used.

An enumerated type is incomplete until after the closing **}**, and there are restrictions on where an incomplete type can be used, based on the size of an object of that type. One situation where the size may be needed is in determining the representation used by an implementation for a pointer to a scalar type (there is no flexibility for pointers to structure and union types). An implementation does not know the minimum storage requirements needed to represent an object having an enumerated type until all of the members of that type had been processed. In the example below, a single pass implementation, that minimizes the storage allocated, and uses different representations for pointers to different scalar types, would not be able to evaluate `sizeof(enum e_T *)` at the point its value is needed to give a value to `e2`.

implementation single pass
implementation single pass
1449 enumerated type incomplete until
footnote 109
pointer to qualified/unqualified types
alignment pointer to structures

```
1  struct s_T;
2  enum e_T {
3             e1=sizeof(struct s_T *),
4             e2=sizeof(enum e_T *),
5            };
6
```

C90

The C90 Standard did not make this observation about implementation behavior.

C++

This behavior is required of a C++ implementation because:

> *The underlying type of an enumeration is an integral type that can represent all the enumerator values defined in the enumeration.*

7.2p5

Common Implementations

Some implementations unconditionally assign the type **int** to all enumerated types. Others assign the integer type with the lowest rank that can represent the values of all of the enumeration constants.

References

1. Gimpel Software. *Reference Manual for PC-lint*. Gimpel Software, 6.00 edition, Jan. 1994.
2. D. M. Jones. The open systems portability checker reference manual. www.knosof.co.uk, 1999.
3. I. Neamtiu. Detailed break-down of general data provided in paper[4] kindly supplied by first author. Jan. 2008.
4. I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, May 2005.
5. P. Research. *QAC Users Guide*. Programming Research, Hersham, Surrey, KT12 5LU, pc version 5.0 edition, Apr. 2003.
6. L. Rosler. The evolution of C–past and future. *AT&T Bell Laboratories Technical Journal*, 63(8):1685–1699, 1984.