The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.4.4.1 Integer constants

integer constant syntax


```
integer-constant:
                decimal-constant integer-suffix_opt
                octal-constant integer-suffix_opt
                hexadecimal-constant integer-suffix_opt
decimal-constant:
                nonzero-digit
                decimal-constant digit
octal-constant:
                0
                octal-constant octal-digit
hexadecimal-constant:
                hexadecimal-prefix hexadecimal-digit
                hexadecimal-constant hexadecimal-digit
hexadecimal-prefix: one of
                0x  0X
nonzero-digit: one of
                1  2  3  4  5  6  7  8  9
octal-digit: one of
                0  1  2  3  4  5  6  7
hexadecimal-digit: one of
                0  1  2  3  4  5  6  7  8  9
                a  b  c  d  e  f
                A  B  C  D  E  F
integer-suffix:
                unsigned-suffix long-suffix_opt
                unsigned-suffix long-long-suffix
                long-suffix unsigned-suffix_opt
                long-long-suffix unsigned-suffix_opt
unsigned-suffix: one of
                u  U
long-suffix: one of
                l  L
long-long-suffix: one of
                ll  LL
```

Commentary

translation phase 7

constant expression syntax

Integer constants are created in translation phase 7 when the preprocessing tokens *pp-number* are converted into tokens denoting various forms of *constant*. *Integer-constant*s always denote positive values. The character sequence `-1` consists of the two tokens `{-}` `{1}`, a constant expression.

An *integer-suffix* can be used to restrict the set of possible types the constant can have, it also specifies the lowest rank an integer constant may have (which for *ll* or *LL* leaves few further possibilities). The *U*, or *u*, suffix indicates that the integer constant is unsigned.

All translation time integer constants are nonnegative. The character sequence `-1` consists of the token sequence unary minus followed by the *decimal-constant* `1`. Support for translation time negative constants in the lexical grammar would create unjustified complexity by requiring lexers to disambiguate binary from unary operators uses in, for instance: `X-1`.

C90

Support for *long-long-suffix* and the nonterminal *hexadecimal-prefix* is new in C99.

C++

The C++ syntax is identical to the C90 syntax.

Support for *long-long-suffix* and the nonterminal *hexadecimal-prefix* is not available in C++.

Common Implementations

Some implementations specify that the suffix *0b* (or *0B*) denotes an integer constant expressed in binary notation. Over the years the C Committee received a number of requests for such a suffix to be added to the C Standard. The Committee did not see sufficient utility for this suffix to be included in C99. The C embedded systems TR specifies *h* and *H* to denote the types **short frac** or **short accum**, and one of *k*, *K*, *r*, and *R* to denote a fixed-point type. Embedded C TR

The IBM ILE C compiler[6] supports a packed decimal data type. The suffix *d* or *D* may be used to specify that a literal has this type. Microsoft C supports the suffixes *i8*, *i16*, *i32*, and *i64* denoting integer constants having the types **byte** (an extension), **short**, **int**, and **__int64**, respectively.

Other Languages

Although Ada supports integer constants having bases between 1 and 36 (e.g., 2#1101 is the binary representation for 10#13), few other languages support the use of suffixes. Ada also supports the use of underscores within an *integer-constant* to make the value more readable.

Coding Guidelines

A study by Brysbaert[1] found that the time taken for a person to process an Arabic integer between 1 and 99 was a function of the logarithm of its magnitude, the frequency of the number (based on various estimates of its frequency of occurrence in everyday life; see Dorogovtsev et al[2] for measurements of numbers appearing in web pages), and sometimes the number of syllables in the spoken form of the value. Subject response times varied from approximately 300 ms for values close to zero, to approximately 550 ms for values in the nineties.

Experience shows that the *long-suffix l* is often visually confused with the *nonzero-digit 1*.[825.1]

Cg 825.1 If a *long-suffix* is required, only the form *L* shall be used.

Cg 825.2 If a *long-long-suffix* is required, only the form *LL* shall be used.

As previously pointed out, constants appearing in the visible form of the source often signify some quantity with real world semantics attached to it. However, uses of the integer constants 0 and 1 in the visible source often have no special semantics associated with their usage. They also represent a significant percentage of the total number of integer constants in the source code (see Figure 825.1). The frequency of occurrence of these values (most RISC processors dedicate a single register to permanently hold the value zero) comes about through commonly seen program operations. These operations include: code to count the number of occurrences of entities, or that contain loops, or index the previous or next element of an array (not that 0 or 1 could not also have similar semantic meaning to other constant values). constant syntax

A blanket requirement that all integer constants be represented in the visible source by symbolic names fails to take into account that a large percentage of the integer constants used in programs have no special meaning associated with them. In particular the integer constants 0 and 1 occur so often (see Figure 825.1) that having to justify why each of them need not be replaced by a symbolic name would have a high cost for an occasional benefit.

Rev 825.3 No integer constant, other than 0 and 1, shall appear in the visible source code, other than as the sole preprocessing token in the body of a macro definition or in an enumeration definition.

[825.1] While the visual similarity between alphabetic letters has been experimentally measured your author is not aware of any experiment that has measured the visually similarity of digits with letters.

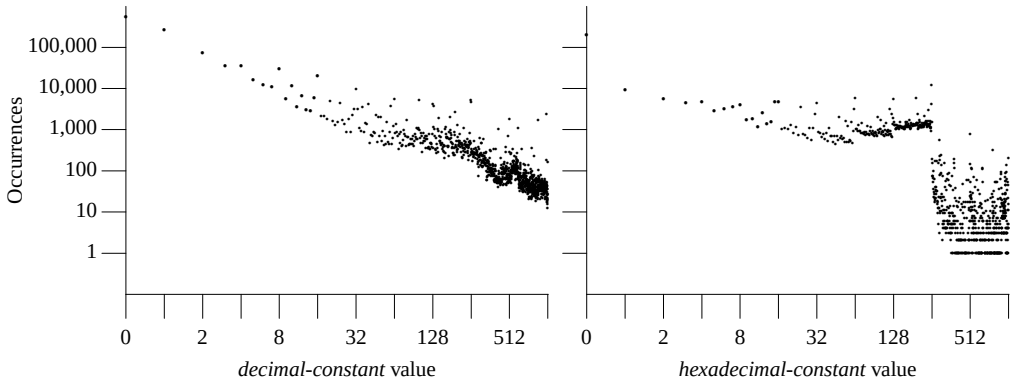

Figure 825.1: Number of integer constants having the lexical form of a *decimal-constant* (the literal 0 is also included in this set) and *hexadecimal-constant* that have a given value. Based on the visible form of the .c and .h files.

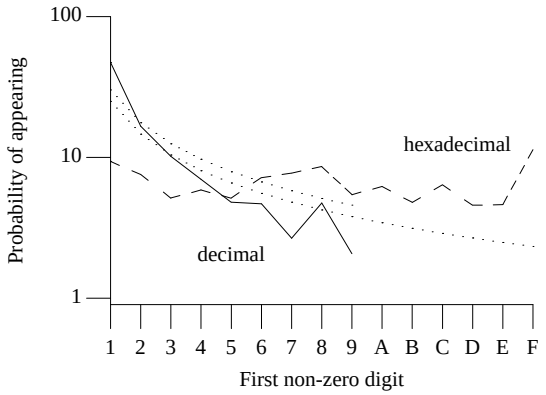

Figure 825.2: Probability of a *decimal-constant* or *hexadecimal-constant* starting with a particular digit; based on .c files. Dotted lines are the probabilities predicted by Benford's law (for values expressed in base 10 and base 16), i.e., $\log(1 + d^{-1})$, where d is the numeric value of the digit.

integer constant with suffix, not immediately converted

Some developers are sloppy in the use of integer constants, using them where a floating constant was the appropriate type. The presence of a period makes it explicitly visible that a floating type is being used. The general issue of integer constant conversions is discussed elsewhere.

Example

pp-number syntax

The character sequence 123xyz is tokenized as {123xyz}, a *pp-number*. This is not a valid integer constant.

Usage

integer constant usage

Having some forms of constant tokens (also see Figure **??**) follow Benford's law[4] would not be surprising because the significant digits of a set of values created by randomly sampling from a variety of different distributions converges to a logarithmic distribution (i.e., Benford's law).[3] While the results for *decimal-constant* (see Figure 825.2) may appear to be a reasonable fit, applying a chi-squared test shows the fit to be remarkably poor (χ^2 = 132,398). The first nonzero digit of *hexadecimal-constant*s appears to be approximately evenly distributed.

Table 825.1: Occurrence of various kinds of *integer-constant*s (as a percentage of all integer constants; note that zero is included in the *decimal-constant* count rather than the *octal-constant* count). Based on the visible form of the `.c` and `.h` files.

Kind of *integer-constant*	`.c` files	`.h` files
decimal-constant	64.1	17.8
hexadecimal-constant	35.8	82.1
octal-constant	0.1	0.2

Table 825.2: Occurrence of various *integer-suffix* sequences (as a percentage of all *integer-constants*). Based on the visible form of the `.c` and `.h` files.

Suffix Character Sequence	`.c` files	`h.` files	Suffix Character Sequence	`.c` files	`.h` files
none	99.6850	99.5997	Lu/lU	0.0005	0.0001
U/u	0.0298	0.0198	LL/lL/ll	0.0072	0.0022
L/l	0.1378	0.2096	ULL/uLl/ulL/Ull	0.0128	0.0061
U/uL/ul	0.1269	0.1625	LLU/lLu/LlU/llu	0.0000	0.0000

Table 825.3: Common token pairs involving *integer-constant*s. Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
, *integer-constant*	42.9	56.5	**(** *integer-constant*	2.8	3.4
integer-constant **]**	6.4	44.4	== *integer-constant*	25.5	2.0
integer-constant ,	58.2	44.2	**return** *integer-constant*	18.6	1.9
integer-constant **;**	14.1	12.1	+ *integer-constant*	33.7	1.9
integer-constant **)**	14.2	11.7	**&** *integer-constant*	30.6	1.5
integer-constant **#**	1.4	9.1	identifier *integer-constant*	0.3	1.5
= *integer-constant*	19.6	9.0	- *integer-constant*	44.0	1.3
[*integer-constant*	39.3	5.6	< *integer-constant*	40.0	1.3
integer-constant **}**	1.2	4.4	**{** *integer-constant*	4.2	1.2
-v *integer-constant*	69.0	4.1			

A study by Pollmann and Jansen[9] analyzed occurrences of related pairs of numerals (e.g., "two or three books") in written (Dutch) text. They found that pairs of numerals often followed what they called *ordering rules*, which were (for the number pair x and y):

- x has to be smaller than y
- x or y has to be *round* (i.e., *round* numbers include the numbers 1 to 20 and the multiples of five)
- the difference between x and y has to be a favorite number. (These include: $10^n \times$(1, 2, ½, or ¼) for any value of n.)

Description

826 An integer constant begins with a digit, but has no period or exponent part. integer constant

Commentary

A restatement of information given in the Syntax clause.

827 It may have a prefix that specifies its base and a suffix that specifies its type.

Commentary

A suffix need not uniquely determine an integer constants type, only the lowest rank it may have. There is no suffix for specifying the type **int**, or any integer type with rank less than **int** (although implementations may provide these as an extension).

base document

The base document did not specify any suffixes; they were introduced in C90.

Other Languages

A few other languages also support some kind of suffix, including C++, Fortran, and Java.

Coding Guidelines

terminology integer constant

Developers do not normally think in terms of an integer constant having a prefix. The term *integer constant* is often used to denote what the standard calls a *decimal constant*, which corresponds to the common case. When they occur in source, both octal and hexadecimal constants are usually referred to by these names, respectively. The benefits of educating developers to use the terminology *decimal constant* instead of *integer constant* are very unlikely to exceed the cost.

---

decimal constant

A decimal constant begins with a nonzero digit and consists of a sequence of decimal digits. 828

Commentary

A restatement of information given in the Syntax clause.

Coding Guidelines

The constant 0 is, technically, an octal constant. Some guideline documents use the term decimal constant in their wording, overlooking the fact that, technically, this excludes the value 0. The guidelines given in this book do not fall into this trap, but anybody who creates a modified version of them needs to watch out for it.

---

octal constant

An octal constant consists of the prefix **0** optionally followed by a sequence of the digits **0** through **7** only. 829

Commentary

A restatement of information given in the Syntax clause. An octal constant is a natural representation to use when the value held in a single byte needs to be displayed (or read in) and the number of output indicators (or input keys) is limited (only eight possibilities are needed). For instance, a freestanding environment where the output device can only represent digits. The users of such input/output devices tend to be technically literate.

Other Languages

A few other languages (e.g., Java and Ada) support octal constants. Most do not.

Common Implementations

K&R C supported the use of the digits 8 and 9 in octal constants (support for this functionality was removed during the early evolution of C[10] although some implementations continue to support it[5, 8]). They represented the values 10 and 11, respectively.

Coding Guidelines

Octal constants are rarely used (approximately 0.1% of all *integer-constant*s, not counting the value 0). There seem to be a number of reasons why developers occasionally use octal constants:

- A long-standing practice that arguments to calls to some Unix library functions use octal constants to indicate various attributes (e.g., `open(file, O_WRONLY, 0666)`). The introduction, by POSIX in 1990, of identifiers representing these properties has not affected many developers' coding habits. The value 0666, in this usage, could be said to be treated like a symbolic identifier.
- Cases where it is sometimes necessary to think of a bit pattern in terms of its numeric value. Bit patterns are invariably grouped into bytes, making hexadecimal an easier representation to manipulate (because its visual representation is easily divisible into bytes and half bytes). However, mental arithmetic

involving octal digits is easier to perform than that with hexadecimal digits. (There are fewer items of information that need to be remembered and people have generally automated the processing of digits, but conscious effort is needed to map the alphabetic letters to their numeric equivalents.)

- The values are copied from an external source; for instance, tables of measurements printed in octal.

There are no obvious reasons for recommending the use of octal constants over decimal or hexadecimal constants (there is a potential advantage to be had from using octal constants).

escape sequence
octal digits

---

830 A hexadecimal constant consists of the prefix **`0x`** or **`0X`** followed by a sequence of the decimal digits and the letters **`a`** (or **`A`**) through **`f`** (or **`F`**) with values 10 through 15 respectively.

hexadecimal constant

Commentary

A restatement of information given in the Syntax clause. A hexadecimal constant provides a natural way of denoting a value, occupying one or more 8-bit bytes, when its underlying representation is of interest. Each digit in a hexadecimal constant represents four binary digits, a nibble.

Other Languages

Many languages support the representation of hexadecimal constants in source code. The prefix character *`$`* (not available in the C basic source character set) is used almost as often, if not more so, than the *`0x`* form of prefix.

Coding Guidelines

In many cases a hexadecimal constant is not thought about, by developers, in terms of being a number but as representing a pattern of bits (perhaps even having an internal structure to it). For instance, in a number of applications objects and constants have values that are more meaningfully thought about in terms of powers of two rather than powers to ten. In such cases a constant appearing in the source as a hexadecimal constant is more easily appreciated (in terms of the sums of the powers of two involved and by which powers of two it differs from other constants) than if expressed as a decimal constant.

Measurements of constant use in source code show that usage patterns for hexadecimal constants are different from decimal constants. The probability of a particular digit being the first nonzero digit in a hexadecimal constant is roughly constant, while the probability distribution of this digit in a decimal constant decreases with increasing value (a ch-squared analysis gives a very low probability of it matching Benford's law). Also the sequence of value digits in a *`hexadecimal-constant`* (see Table 830.1) almost always exactly corresponds to the number of nibbles in either a character type, **`short`**, **`int`**, or **`long`**.

825 integer constant usage

A study by Logan and Klapp[7] used *alphabet arithmetic* (e.g., $A + 2 = C$) to investigate how extended practice and rote memorization affected automaticity. For inexperienced subjects who had not memorized any addition table, the results showed that the time taken to perform the addition increased linearly with the value of the digit being added. This is consistent with subjects counting through the letters of the alphabet to obtain the answer. With sufficient practice subjects performance not only improved but became digit-independent. This is consistent with subjects recalling the answer from memory; the task had become automatic.

automatization

The *practice* group of subjects were given a sum and had to produce the answer. The *memorization* group of subjects were asked to memorise a table of sums and there answers, (e.g., $A + 2 = C$). In both cases the results showed that performance was proportional to the number of times each question/answer pair had been encountered, not the total amount of time spent.

Arithmetic involving hexadecimal constants differs from that involving decimal constants in that developers will have had much less experience in performing it. The results of the Logan and Klapp study show that the only way for developers to achieve the same level of proficiency is to commit the hexadecimal addition table to memory. Whether the cost of this time investment has a worthwhile benefit is unknown.

Table 830.1: Occurrence of *hexadecimal-constant*s containing a given number of digits (as a percentage of all such constants). Based on the visible form of the `.c` files.

Digits	Occurrence	Digits	Occurrence	Digits	Occurrence	Digits	Occurrence
0	0.003	5	0.467	10	0.005	15	0.000
1	1.092	6	0.226	11	0.001	16	0.209
2	59.406	7	0.061	12	0.001		
3	1.157	8	2.912	13	0.000		
4	34.449	9	0.010	14	0.000		

Semantics

The value of a decimal constant is computed base 10; 831

Commentary

C supports the representation of constants in the base chosen by evolution on planet Earth.

that of an octal constant, base 8; 832

unsigned integer types
object representation

Commentary

The C language requires the use of binary representation for the integer types. The use of both base 8 and base 16 visual representations of binary information has been found to be generally more efficient, for people, than using a binary representation. Developers continue to debate the merits of one base over another. Both experience with using one particular base and the kind of application domain affect preferences.

that of a hexadecimal constant, base 16. 833

Commentary

The correct Latin prefix is *sex*, giving *sexadecimal*. It has been claimed that this term was considered too racey by IBM who adopted *hexadecimal* (*hex* is the equivalent Greek prefix, the Latin *decimal* being retained) in the 1960s to replace it (the term was used in 1952 by Carl-Eric Froeberg in a set of conversion tables).

The lexically first digit is the most significant. 834

Commentary

The Arabic digits in a constant could be read in any order. In Arabic, words and digits are read/written right-to-left (least significant to most significant in the case of numbers). The order in which Arabic numerals are written was exactly copied by medieval scholars, except that they interpreted them using the left-to-right order used in European languages.

integer constant type first in list

The type of an integer constant is the first of the corresponding list in which its value can be represented. 835

Commentary

translation phase 7

This list only applies to those *pp-numbers* that are converted to *integer-constant* tokens as part of translation phase 7. Integer constants in **`#if`** preprocessor directives always have type `intmax_t`, or `uintmax_t` (in C90 they had type **`long`** or **`unsigned long`**).

Other Languages

In Java integer constants have type **`int`** unless they are suffixed with *l*, or *L*, in which case they have type **`long`**. Many languages have a single integer type, which is also the type of all integer constants.

Coding Guidelines

The type of an integer constant may depend on the characteristics of the host on which the program executes and the form used to express its value. For instance, the integer constant 40000 may have type **int** or **long int** (depending on whether **int** is represented in more than 16 bits, or in just 16 bits). The hexadecimal constant 0x9C40 (40000 decimal) may have type **int** or **unsigned int** (depending on the whether **int** is represented in more than 16 bits, or in just 16 bits).

For objects having an integer type there is a guideline recommending that a single integer type always be used (the type **int**). However, integer constants never have a type whose rank is less than **int** and so the developer issues associated with the integer promotions do not apply. It makes no sense for a coding guideline to recommend against the use of an *integer-constant* whose value is not representable in the type **int** (a developer is unlikely to use such a value without the application requiring it).

?? object
int type only

The possibility that the type of an integer constant can vary between implementations and platforms creates a portability cost. There is also the potential for incorrect developer assumptions about the type of an integer constant, leading to additional maintenance costs. The specification of a guideline recommendation is complicated by the fact that C does not support a suffix that specifies the type **int** (or its corresponding unsigned version). This means it is not possible to specify that a constant, such as 40000, has type **int** and expect a diagnostic to appear when using a translator that gives it the type **long**.

Cg 835.1 An unsuffixed *integer-constant* having a value greater than 32767 shall be treated, for the purposes of these guideline recommendations, as if its lexical included a suffix specifying the type **int**.

An integer constant containing a suffix is generally taken as a statement of intent by the developer. A suffixed integer constant that is immediately converted to another type is suspicious.

Cg 835.2 An integer constant containing a suffix shall not be immediately converted to another type.

Dev 835.2 The use of a macro defined in a system header may be immediately cast to another type.

Dev 835.2 The use of a macro defined in a developer written system header may be immediately cast to another type, independent of how the macro is implemented.

Dev 835.2 The body of a macro may convert, to an integer type, one of the parameters of that macro definition.

Dev 835.2 If the range of values supported by the type **unsigned short**, or **unsigned char**, is the same as that supported by **unsigned int**, an integer constant containing an unsigned suffix may be converted to those types.

Is there anything to be gained from recommending that integer constants less than 32767 be suffixed rather than implicitly converted to another type? The original type of such an integer constant is obvious to the reader and a conversion to a type for which the standard provides a suffix will not change its value; the real issue is developer expectation. Expectation can become involved through the semantics of what the constant represents. For instance, a program that manipulates values associated with the ISO 10646 Standard may store these values in objects that always have type **unsigned int**. This usage can lead to developers learning (implicitly or explicitly) that objects manipulating these semantic quantities have type **unsigned int**, creating an expectation that all such quantities have this type. Expectations on the sign of an operand can show up as a difference between actual and expected behavior; for instance, the following expression checks if any bits outside of the least significant octet are set: `~F00_char > 0x00ff`. It only works if the left operand has an unsigned type. (If it has a signed type, setting the most significant bit will cause the result to be negative.) If the identifier `F00_char` is a macro whose body is a constant integer having a signed type, developer expectations will not have been met.

implicit learning

In those cases where developers have expectations of an operand having a particular type, use of a suffix can help ensure that this expectation is met. If the integer constant appears in the visible source at the point its value is used, developers can immediately deduce its type. An integer constant in the body of a macro definition or as an argument in a macro invocation are the two circumstances where type information is not immediately apparent to readers of the source. (The integer constant is likely to be widely separated from its point of use in an expression.)

The disadvantage of specifying a suffix on an integer constant because of the context in which it is used is that the applicable type may change. The issues involved with implicit conversion versus explicit conversion are discussed elsewhere. An explicit cast, using a typedef name rather than a suffix, is more flexible in this regard.

implicit conversion

Use of a suffix not defined by the standard, but provided by the implementation, is making use of an extension. Does this usage fall within the guideline recommendation dealing with use of extensions, or is it sufficiently useful that a deviation should be made for it? Suffixes are a means for the developer to specify type information on integer constants. Any construct that enables the developer to provide more information is usually to be encouraged. While there are advantages to this usage, at the time of this writing insufficient experience is available on the use of suffixes to know whether the advantages outweigh the disadvantages. A deviation against the guideline recommendation might be applicable in some cases.

extensions ?? cost/benefit

Dev ?? Any integer constant suffix supported by an implementation may be used.

Table 835.1: Occurrence of *integer-constant*s having a particular type (as a percentage of all such constants; with the type denoted by any suffix taken into account) when using two possible representations of the type **int** (i.e., 16- and 32-bit). Based on the visible form of the `.c` and `.h` files.

Type	16-bit **int**	32-bit **int**
int	94.117	99.271
unsigned int	3.493	0.414
long	1.805	0.118
unsigned long	0.557	0.138
other-types	0.029	0.059

integer constant possible types

836

Suffix	Decimal Constant	Octal or Hexadecimal Constant
none	**int** **long int** **long long int**	**int** **unsigned int** **long int** **unsigned long int** **long long int** **unsigned long long int**
u or **U**	**unsigned int** **unsigned long int** **unsigned long long int**	**unsigned int** **unsigned long int** **unsigned long long int**
l or **L**	**long int** **long long int**	**long int** **unsigned long int** **long long int** **unsigned long long int**
Both **u** or **U** and **l** or **L**	**unsigned long int** **unsigned long long int**	**unsigned long int** **unsigned long long int**
ll or **LL**	**long long int**	**long long int** **unsigned long long int**
Both **u** or **U** and **ll** or **LL**	**unsigned long long int**	**unsigned long long int**

Commentary

The lowest rank that an integer constant can have is type **int**. This list contains the standard integer types only, giving preference to these types. Any supported extended integer type is considered if an appropriate type is not found from this list.

C90

The type of an integer constant is the first of the corresponding list in which its value can be represented. Unsuffixed decimal: ***int***, ***long int***, ***unsigned long int****; unsuffixed octal or hexadecimal:* ***int***, ***unsigned int***, ***long int***, ***unsigned long int****; suffixed by the letter* `u` *or* `U`*:* ***unsigned int***, ***unsigned long int****; suffixed by the letter* `l` *or* `L`*:* ***long int***, ***unsigned long int****; suffixed by both the letters* `u` *or* `U` *and* `l` *or* `L`*:* ***unsigned long int****.*

Support for the type **long long** is new in C99.

The C90 Standard will give a sufficiently large decimal constant, which does not contain a *u* or *U* suffix— the type **unsigned long**. The C99 Standard will never give a decimal constant that does not contain either of these suffixes— an unsigned type.

Because of the behavior of C++, the sequencing of some types on this list has changed from C90. The following shows the entries for the C90 Standard that have changed.

Suffix	Decimal Constant
none	**int** **long int** **unsigned long int**
`l` or `L`	**long int** **unsigned long int**

Under C99, the none suffix, and *l* or *L* suffix, case no longer contain an unsigned type on their list.

A decimal constant, unless given a *u* or *U* suffix, is always treated as a signed type.

C++

If it is decimal and has no suffix, it has the first of these types in which its value can be represented: ***int***, ***long int****; if the value cannot be represented as a* ***long int****, the behavior is undefined. If it is octal or hexadecimal and has no suffix, it has the first of these types in which its value can be represented:* ***int***, ***unsigned int***, ***long int***, ***unsigned long int****. If it is suffixed by* `u` *or* `U`*, its type is the first of these types in which its value can be represented:* ***unsigned int***, ***unsigned long int****. If it is suffixed by* `l` *or* `L`*, its type is the first of these types in which its value can be represented:* ***long int***, ***unsigned long int****. If it is suffixed by* `ul`, `lu`, `uL`, `Lu`, `Ul`, `lU`, `UL`, *or* `LU`*, its type is* ***unsigned long int****.* 2.13.1p2

The C++ Standard follows the C99 convention of maintaining a decimal constant as a signed and never an unsigned type.

The type **long long**, and its unsigned partner, is not available in C++.

There is a difference between C90 and C++ in that the C90 Standard can give a sufficiently large decimal literal that does not contain a *u* or *U* suffix— the type **unsigned long**. Neither the C++ or C99 Standard will give a decimal constant that does not contain either of these suffixes— an unsigned type.

Other Languages

In Java hexadecimal and octal literals always have a signed type and denote a negative value if the high-order bit, for their type, is set. The literal `0xcafebabe` has decimal value -889275714 and type **int** in Java, and decimal value 3405691582 and type **unsigned int** or **unsigned long** in C.

If an integer constant cannot be represented by any type in its list, it may have an extended integer type, if the extended integer type can represent its value. 837

Commentary

For an implementation to support an integer constant which is not representable by any standard integer type, requires that it support an extended integer type that can represent a greater range of values than the types **`long long`** or **`unsigned long long`**.

C90

Explicit support for extended types is new in C99.

C++

The C++ Standard allows new object types to be created. It does not specify any mechanism for giving literals these types.

A C translation unit that contains an integer constant that has an extended integer type may not be accepted by a conforming C++ translator. But then it may not be accepted by another conforming C translator either. Support for the construct is implementation-defined.

Other Languages

Very few languages explicitly specify potential implementation support for extended integer types.

Common Implementations

integer 825 constant syntax

In some implementations it is possible for an integer constant to have a type with lower rank than those given on this list.

Coding Guidelines

extensions ?? cost/benefit

Source containing an integer constant, the value of which is not representable in one of the standard integer types, is making use of an extension. The guideline recommendation dealing with use of extensions is applicable here. If it is necessary for a program to use an integer constant having an extended integer type, the deviation for this guideline specifies how this usage should be handled. The issue of an integer constant being within the range supported by a standard integer type on one implementation and not within range on another implementation is discussed elsewhere.

integer 835.1 constant greater than 32767

If all of the types in the list for the constant are signed, the extended integer type shall be signed. 838

Commentary

This is a requirement on the implementation. This requirement applies to the standard integer types. By requiring that any extended integer type follow the same rule, the standard is preserving the idea that decimal constants are signed unless they contain an unsigned suffix. All of the types in the list are signed if the lexical representation is a decimal constant without a suffix, or a decimal constant whose suffix is not *u* or *U*.

If all of the types in the list for the constant are unsigned, the extended integer type shall be unsigned. 839

Commentary

This is a requirement on the implementation. The types in the list are all unsigned if the integer constant contains a *u* or *U* suffix.

If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned. 840

Commentary

Both signed and unsigned types only occur if octal or hexadecimal notation is used, and no *u* or *U* suffix appears in the constant. There is no requirement on the implementation to follow the signed/unsigned pattern seen for the standard integer types when octal and hexadecimal notation is used for the constants.

integer constant
no type

841 If an integer constant cannot be represented by any type in its list and has no extended integer type, then the integer constant has no type.

Commentary

Consider the token 100000000000000000000 in an implementation that supports a 64-bit two's complement **`long long`**, and no extended integer types. The numeric value of this token outside of the range of any integer type supported by the implementation and therefore it has no type.

This sentence was added by the response to DR #298.

References

1. M. Brysbaert. Arabic number reading: On the nature of the numerical scale and the origin of phonological recoding. *Journal of Experimental Psychology: General*, 124(4):434–452, 1995.
2. S. N. Dorogovtsev, J. F. F. Mendes, and J. G. Oliveira. Frequency of occurrence of numbers in the World Wide Web. *Physica A*, 360(2):548–556, 2006.
3. T. P. Hill. A statistical derivation of the significant-digit law. *Statistical Science*, 10:354–363, 1996.
4. T. P. Hill. The first-digit phenomenon. *American Scientist*, 86:358–363, July-Aug. 1998.
5. HP. *DEC C Language Reference Manual*. Compaq Computer Corporation, aa-rh9na-te edition, July 1999.
6. IBM. *WebSphere Development Studio ILE C/C++ Programmer's Guide*. IBM Canada Ltd, Ontario, Canada, sc09-27 12-02 edition, May 2001.
7. G. D. Logan and S. T. Klapp. Automatizing alphabet arithmetic: I. Is extended practice necessary or produce automaticity? *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 17(2):179–195, 1991.
8. Perkin-Elmer. *C PROGRAMMING Manual*. Perkin-Elmer Corporation, Oceanport, New Jersey 07757, 1984.
9. T. Pollmann and C. Jansen. The language user as an arithmetician. *Cognition*, 59:219–237, 1996.
10. L. Rosler. The evolution of C–past and future. *AT&T Bell Laboratories Technical Journal*, 63(8):1685–1699, 1984.