

# **The New C Standard** (Excerpted material)

---

**An Economic and Cultural Commentary**

**Derek M. Jones**

derek@knosof.co.uk

## 6.4.4.4 Character constants

charac-  
ter constant  
syntax  
escape sequence  
syntax

---

```

character-constant:
    ' c-char-sequence '
    L' c-char-sequence '

c-char-sequence:
    c-char
    c-char-sequence c-char

c-char:
    any member of the source character set except
        the single-quote ' , backslash \ , or new-line character
    escape-sequence

escape-sequence:
    simple-escape-sequence
    octal-escape-sequence
    hexadecimal-escape-sequence
    universal-character-name

simple-escape-sequence: one of
    \ ' \ " \ ? \ \
    \ a \ b \ f \ n \ r \ t \ v

octal-escape-sequence:
    \ octal-digit
    \ octal-digit octal-digit
    \ octal-digit octal-digit octal-digit

hexadecimal-escape-sequence:
    \ x hexadecimal-digit
    hexadecimal-escape-sequence hexadecimal-digit

```

### Commentary

character  
or " matches

The following is an undefined behavior potentially leading to a syntax violation:

```
1 char c = '\z';
```

footnote<sup>879</sup>  
65

because the characters that may follow a backslash are enumerated by the syntax. A backslash must be followed by one of the characters listed in the C sentence. This character sequence tokenizes as:

```
{char} {c} {=} {'}{\}\{z}{'}{;}{;}
```

character<sup>886</sup>  
constant  
single char-  
acter value

A character-constant, that does not contain more than one character, is effectively preceded by an implicit conversion to the type **char**. If this type is signed and the constant sufficiently large, the resulting value is likely to be negative (the most common, undefined behavior, being to wrap).

Rationale

Proposals to add '\e' for ASCII ESC ('\033') were not adopted because other popular character sets have no obvious equivalent.

Including *universal-character-name* as an *escape-sequence* removes the need to explicitly include it in other wording that refers to escape sequences (e.g., in conditional inclusion).

Lowercase letters as escape sequences are also reserved for future standardization, although permission is given to use other letters in extensions.

### C90

Support for *universal-character-name* is new in C99.

**C++**

The C++ Standard classifies *universal-character-name* as an *escape-sequence*, not as a *c-char*. This makes no difference in practice to the handling of such *c-chars*.

**Other Languages**

In other languages the sequence of characters within a character constant usually forms part of the lexical grammar, not the language syntax. The constructs supported by *simple-escape-sequence* was once unique to C. A growing number of subsequent language (e.g., C++, C#, and Java; although Java does not support `\?`, `\a`, `\v`, or hexadecimal escape sequences). Perl also includes support for the escape sequences `\e` (escape) and `\cC` (Control-C).

**Common Implementations**

While lexing a character constant most translators simply look for a closing single-quote to match the one that began the *character-constant*. The internal, syntactic structure of the characters constant is not usually examined until a later phase of translation. As a consequence, many implementations do not diagnose the previous example as a violation of syntax.

**Coding Guidelines**

Character constants are usually used in code that processes data consisting of sequences of characters. This data may be, for instance, a line of text read from a file or command options typed as input to an executing program. Character constants are not usually the operands of arithmetic operators (see Table 866.3) or the index of an array (581 instances, or 1.4% of all character constants, versus 90,873 instances, or 5.3% of all integer constants).

Does the guideline recommendation dealing with giving symbolic names to constants unconditionally apply to character constants? The reasons given for why symbolic names should be used in preference to the constant itself include:

- *The value of the constant is likely to be changed during program maintenance.* Experience shows that the characters of interest to programs that process sequences of characters do not change frequently. Programs tend to process sets of characters (e.g., alphabetic characters), as shown by the high percentage of characters used in case labels (see Table 866.3).
- *A more meaningful semantic association is created.* Developers are experienced readers of sequences of characters (text) and they have existing strong semantic associations with the properties of many characters; for instance, it is generally known that the space character is used to separate words. That is, giving the name `word_delimiter` to the character constant `' '` is unlikely to increase the amount of semantic association.
- *The number of cognitive switches a reader has to perform is reduced.* Most of the characters constants used (see Figure 884.1) have printable glyphs. Are more or fewer cognitive switches needed to comprehend that, for instance, either `word_delimiter` or `' '` is a character used to delimit words? Use of the escape sequence `'\x20'` would require readers to make several changes of mental representation.

?? integer  
constant  
not in visible  
source  
?? floating  
constant  
not in visible  
source  
constant  
syntax

cognitive  
switch

The distribution of characters constants in the visible source (see Figure 884.1) has a different pattern from that of integer constants (see Figure ??). There are also more character constants whose lexical form might be said to provide sufficient symbolic information (e.g., the null character, new-line, space, the digit zero, etc.).

The number of cases where the appearance of a symbolic name, in the visible source, is more cost effective than a character constant would appear to be small. Given the cost of checking all occurrences of character constants for the few where replacement by a symbolic name would provide a benefit, a guideline recommendation is not considered worthwhile.

**Table 866.1:** Occurrence of various kinds of *character-constant* (as a percentage of all such constants). Based on the visible form of the `.c` files.

Kind of <i>character-constant</i>	% of all <i>character-constants</i>
not an escape sequence	76.1
<i>simple-escape-sequence</i>	8.8
<i>octal-escape-sequence</i>	15.1
<i>hexadecimal-escape-sequence</i>	0.0
<i>universal-character-name</i>	0.0

**Table 866.2:** Occurrence of *escape-sequences* within *character-constants* and *string-literals* (as a percentage of *escape-sequences* for that kind of token). Based on the visible form of the `.c` files.

Escape Sequence	% of <i>character-constant</i> Escape Sequences	% of <i>string-literal</i> escape sequences	Escape sequence	% of <i>character-constant</i> Escape Sequences	% of <i>string-literal</i> Escape Sequences
<code>\n</code>	18.10	79.15	<code>\b</code>	0.66	0.04
<code>\t</code>	3.90	11.62	<code>\'</code>	3.24	0.02
<code>\"</code>	1.29	3.08	<code>\%</code>	0.00	0.02
<code>\0</code>	52.70	2.06	<code>\v</code>	0.31	0.01
<code>\x</code>	0.12	1.10	<code>\p</code>	0.00	0.01
<code>\2</code>	2.73	1.01	<code>\f</code>	0.44	0.01
<code>\\</code>	5.70	0.61	<code>\?</code>	0.01	0.01
<code>\r</code>	3.01	0.46	<code>\e</code>	0.00	0.00
<code>\3</code>	4.95	0.42	<code>\a</code>	0.11	0.00
<code>\1</code>	2.72	0.35			

**Table 866.3:** Common token pairs involving *character-constants*. 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
<code>== character-constant</code>	7.1	22.8	<code>character-constant   </code>	4.2	4.2
<code>, character-constant</code>	0.3	18.1	<code>character-constant &amp;&amp;</code>	5.3	3.3
<code>case character-constant</code>	8.5	16.7	<code>&lt;= character-constant</code>	7.1	1.7
<code>= character-constant</code>	0.8	14.2	<code>&gt;= character-constant</code>	3.6	1.5
<code>!= character-constant</code>	5.3	8.4	<code>character-constant )</code>	33.0	0.7
<code>( character-constant</code>	0.1	6.1	<code>character-constant ,</code>	17.6	0.3
<code>character-constant :</code>	16.7	6.0	<code>character-constant ;</code>	16.6	0.3

## Description

integer character constant

An integer character constant is a sequence of one or more multibyte characters enclosed in single-quotes, as in `'x'`. 867

## Commentary

The syntax specification does not use the term *integer character constant*. It is used here, and throughout the standard, to distinguish character constants that are not wide character constants. While a character constant has type `int`, the common developer terminology is still *character constant* (the wide form being relatively rare; its use is explicitly called out).

## C90

The example of `ab` as an integer character constant has been removed from the C99 description.

C++

*A character literal is one or more characters enclosed in single quotes, as in 'x', . . .*

2.13.2p1

A multibyte character is replaced by a *universal-character-name* in C++ translation phase 1. So, the C++ Standard does not need to refer to such entities here.

### Coding Guidelines

The phrase *integer character constant* is rarely heard, although it is descriptive of the type and form of this integer constant. The common usage terminology is *character constant*, which does suggest that the constant has a character type. Coding guideline documents need to use the term *integer character constant* to remind developers that this form of constant has type **int**.

868 A wide character constant is the same, except prefixed by the letter L.

### Commentary

The term *wide character constant* is used by developers to describe this kind of character constant. The prefix is an uppercase *L* only. There is no support for use of a lowercase letter.

### Other Languages

Support for some form of wide characters is gradually becoming more generally available in programming languages. Fortran (since the 1991 standard) supports *char-literal-constants* that contain a prefix denoting the character set used (e.g., NIHONGO\_ 'some kanji here'). Later versions of Cobol supported similar functionality.

### Common Implementations

While all implementations are required to support this form of character constant, they can vary significantly in the handling of the character sequences appearing within single-quotes. Many implementations simply support the same set of characters, in this context, as when no L prefix is given.

869 With a few exceptions detailed later, the elements of the sequence are any members of the source character set;

### Commentary

The exceptions can occur through the use of escape sequences. The elements may also denote characters that are not in the source character set if an implementation supports any.

C++

*[Note: in translation phase 1, a universal-character-name is introduced whenever an actual extended character is encountered in the source text. Therefore, all extended characters are described in terms of universal-character-names. However, the actual compiler implementation may use its own native character set, so long as the same results are obtained. ]*

2.13.2p5

In C++ all elements in the sequence are characters in the source character set after translation phase 1. The creation of *character-literal* preprocessing tokens occurs in translation phase 3, rendering this statement not applicable to C++.

### Common Implementations

Most implementations support characters other than members of the source character set. Any character that can be entered into the source code, using an editor, are usually supported within a character constant.

character  
constant  
mapped

translation phase  
4  
translation phase  
5  
translation phase  
1  
footnote  
141

they are mapped in an implementation-defined manner to members of the execution character set.

### Commentary

This mapping can either occur in translation phase 4 (when the character constant occurs in an expression within a preprocessing directive) or 5 (all other occurrences). Those preprocessing tokens that are part of a preprocessing directive will have the values given them in translation phase 1. The two mapped values need not be the same.

**C++**

2.13.2p1 *An ordinary character literal that contains a single `c-char` has type **char**, with value equal to the numerical value of the encoding of the `c-char` in the execution character set.*

2.2p3 *The values of the members of the execution character sets are implementation-defined, . . .*

2.13.2p2 *The value of a wide-character literal containing a single `c-char` has value equal to the numerical value of the encoding of the `c-char` in the execution wide-character set.*

Taken together, these statements have the same meaning as the C specification.

escape se-  
quences  
character con-  
stant  
character  
constant  
escape se-  
quences

The single-quote `'`, the double-quote `"`, the question-mark `?`, the backslash `\`, and arbitrary integer values are representable according to the following table of escape sequences: 871

single quote	<code>'</code>	<code>\'</code>
double quote	<code>"</code>	<code>\"</code>
question mark	<code>?</code>	<code>\?</code>
backslash	<code>\</code>	<code>\\</code>
octal character		<code>\octal digits</code>
hexadecimal character		<code>\xhexadecimal digits</code>

### Commentary

The standard defines a set of escape sequences that can occur both in character constants and string literals. It does not define a separate set for each kind of constant. A consequence of this single set is that there are multiple representations for some characters, for the two kinds of constants. Being able to prefix both the `'` and `"` characters with a backslash, without needing to know what delimiter they appear between, simplifies the automatic generation of C source code.

The single-quote `'` and the double-quote `"` characters have special meaning within character constants and string literals. The question-mark `?` character can have special meaning as part of a trigraph. To denote these characters in source code, some form of escape sequence is required, which in turn adds the escape sequence character (the backslash `\` character) to the list of special characters. Preceding any of these special characters with a backslash character is the convention used to indicate that the characters represent themselves, not their special meaning.

Octal and hexadecimal escape sequences provide a mechanism for developers to specify the numeric value of individual execution-time characters within an integer character constant. While the syntax does permit an arbitrary number of digits to occur in a hexadecimal escape sequence, the range of values of an integer character constant cannot be arbitrarily large.

The conversion of these escape sequences occurs in translation phase 5.

escape se-  
quence  
value within range  
translation phase  
5

**C++**

The C++ wording, in Clause 2.13.2p3, does discuss arbitrary integer values and the associated Table 5 includes all of the defined escape sequences.

**Other Languages**

A number of languages designed since C have followed its example in offering support for escape sequences. Java does not support the escape sequence `\?`, or hexadecimal characters.

**Common Implementations**

Some implementations treat a backslash character followed by any character not in the preceding list as representing the following character only.

**Coding Guidelines**

Experience has shown that it is easy for readers to be confused by the character sequences `\'` and `\\` when they occur among other integer character constants. However, a guideline recommendation telling developers to be careful serves no useful purpose.

What purpose does having a character constant containing an octal or hexadecimal escape sequence in the source serve? Since the type of an integer character constant is **int**, not **char**, an integer constant with the same value could appear in every context that an integer character constant could appear, with no loss of character set independence (because there was none in the first place).

One reason for using quotes, even around a hexadecimal or octal escape sequence, is to maintain the semantic associations, in a readers head, of dealing with a character value. Although integer character constants have type **int**, experience suggests that developers tend to think of them in terms of a character type. When comprehending code dealing with objects having character types, an initializer, for instance, containing other character constants, or values having semantic associations with character sets, a reader's current frame of mind is likely to be character based not integer based. The presence of a value between single-quotes is perhaps sufficient to maintain a frame of mind associated with character, not integer, values (i.e., there are no costs associated with a cognitive switch).

cognitive  
switch

Another reason for use of an escape sequence in a character constant is portability to C++, where the type of an integer character constant is **char**, not **int**. In this case use of an integer constant would not be equivalent (particularly if the constant was an argument to an overloaded function).

The character sequence `\?` is needed when sequences of the `?` character occur together and a trigraph is not intended.

trigraph  
sequences  
replaced by

**Example**

```

1  #include <stdio.h>
2
3  #define TRUE  ('/'/'/')
4  #define FALSE ('-'-'-'')
5
6  int main(void)
7  {
8  printf("%d%s\n", '\',','); // confusing ");
9  }
```

872 The double-quote `"` and question-mark `?` are representable either by themselves or by the escape sequences `\"` and `\?`, respectively, but the single-quote `'` and the backslash `\` shall be represented, respectively, by the escape sequences `\'` and `\\`.

**Commentary**

The sequence `\\` is only mapped once; that is, the sequence `\\\\` represents the two characters `\\`, not the single character `\`.

## Other Languages

To represent the character constant delimiter Pascal uses the convention of two delimiters immediately adjacent to each other (e.g., `''` represents the character single-quote).

escape sequence  
octal digits

The octal digits that follow the backslash in an octal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant.

873

### Commentary

The value of the octal digits is taken as representing the value of a mapped single character in the execution character set. It is not possible to use an octal escape sequence to represent a character whose value is greater than what can be represented within three octal digits (e.g., 511). For the most common case of `CHAR_BIT` having a value of eight, this is not a significant limitation.

`CHAR_BIT`  
macro

escape sequence  
octal value

The numerical value of the octal integer so formed specifies the value of the desired character or wide character.

874

### Commentary

The maximum value that can be represented by three octal digits is 511. There are no prohibitions on using any of the octal values within the range of values that can be represented (unlike use of the `\u` form).

universal  
character  
name  
syntax

### Common Implementations

Most implementations use an 8-bit byte, well within the representable range of an octal escape sequence.

### Coding Guidelines

As pointed out elsewhere, a character constant is likely to be more closely associated in a developer's mind with characters rather than integer values. One reason for representing characters in the execution character set using escape sequences is that a single character representation may not be available in the source character set. Is there any reason for a character constant to contain an octal escape sequence whose value represents a member of the execution character set that is representable in the source character set? There are two possible reasons:

1. For wide character constants, the mapping to the execution character might not be straight-forward. Use of escape sequences may be the natural notation to use to represent members. In this case it is possible that an escape sequence just happens to have the same value as a character in the source character set.
2. The available keyboards may not always support the required character (trigraphs are only available for characters in the basic source character set).

basic source  
character set

### Example

```
1 char CHAR_EOT = '\004';
2 char CHAR_DOL = '\044';
3 char CHAR_A   = '\101';
```

escape sequence  
hexadecimal  
digits

The hexadecimal digits that follow the backslash and the letter `x` in a hexadecimal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant.

875

## Commentary

A hexadecimal escape sequence can denote a value that is outside the range of values representable in an implementation's `char` or `wchar_t` type. However, the specification states that an escape sequence represents a single character. This means that a large value cannot be taken to represent two or more characters (in an implementation that supports more than one character in an integer character constant) by using a hexadecimal value that represents the combined value of those two mapped characters.

## Coding Guidelines

Occurrences of hexadecimal escape sequences are very rare in the visible source of the `.c` files. Given this rarity, developers are much more likely to be unfamiliar with the range of possible behaviors of hexadecimal escape sequences than the behaviors of octal escape sequences. Possible unexpected behaviors include:

- The character `0` appears before the `x` or `X` in a hexadecimal constant, but not between the backslash and the `x` in an escape sequence.
- An arbitrary number of digits are permitted in a hexadecimal escape sequence, unlike octal escape sequences that contain a maximum of three digits. Padding the numeric part of hexadecimal escape sequences with leading zeros can provide a misleading impression. It is possible for a character not intended, by the developer, to be part of the hexadecimal escape sequence to be treated, by a translator, as being part of that sequence. (This requires implementation support for more than one character in a character constant.) This situation is more likely to occur in string literals.

866 escape sequence syntax

873 escape sequence octal digits

877 escape sequence longest character sequence

885 character constant more than one character string literal syntax

Cg 875.1

A hexadecimal escape sequence shall not be used in an integer character constant.

## Example

```
1 char ch_1 = '\x0000000000000000000000000041';
2 char ch_2 = '\0x42';
```

876 The numerical value of the hexadecimal integer so formed specifies the value of the desired character or wide character.

escape sequence hexadecimal value

## Commentary

While there is no restriction on the number of digits in a hexadecimal escape sequence, there is a requirement that the value be in the range representable by the type `unsigned char` (which on some implementations is 32 bits) or the type `wchar_t`. There are no prohibitions on using any of the hexadecimal values within the range of values that can be represented (unlike use of the `\u` form).

882 escape sequence value within range

universal character name syntax

## Common Implementations

The *desired character* in the execution environment, that is. As far as most translators are concerned, the numerical value is a bit pattern. In general they have no knowledge of which, if any, character this represents in the execution character set.

877 Each octal or hexadecimal escape sequence is the longest sequence of characters that can constitute the escape sequence.

escape sequence longest character sequence

## Commentary

This specification resolves an ambiguity in the syntax. Although it is a little misleading in that it can be read to suggest that both octal and hexadecimal escape sequences may consist of an arbitrary long sequence of characters, the syntax permits a maximum of three digit characters in an octal escape sequence.

### Coding Guidelines

Character constants usually consist of a single character, so much of the following discussion is not going to be applicable to them as often as it is to string literals.

For both forms of escape sequence there is the danger that any applicable digit characters immediately following them will be taken to be part of that sequence. The possible number of digits in an octal escape sequence is limited to three. If this number of digits is always used, there is never the possibility of a following character being included in the sequence.

Cg 877.1

An octal escape sequence shall always contain three digits.

Dev 877.1

The lexical form `'\0'` may be used to represent the null character.

CHAR\_BIT  
macro

If CHAR\_BIT has a value greater than 9, an octal escape sequence will not be able to denote all of the representable values in the type **unsigned char**.

Dev 875.1

A hexadecimal escape sequence may be used if an octal escape sequence cannot represent the required value.

### Example

```
1 char a_char = '\02';
2 char null_char = '\0';
3 char b_char = '\0012';
```

---

In addition, characters not in the basic character set are representable by universal character names and certain nongraphic characters are representable by escape sequences consisting of the backslash `\` followed by a lowercase letter: `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, and `\v`.<sup>65)</sup> 878

### Commentary

A restatement of information given in the Syntax clause. The characters are nongraphical in the sense that they do not represent a glyph corresponding to a printable character. Their semantics are discussed elsewhere.

### C90

Support for universal character names is new in C99.

### C++

Apart from the rule of syntax given in Clause 2.13.2 and Table 5, there is no other discussion of these escape sequences in the C++ Standard. :-O

### Other Languages

Support for universal character names is slowly being added to other ISO standardized languages. Java contains similar escape sequences.

### Common Implementations

Some implementations define the escape sequence `\e` to denote *escape*; the Perkin-Elmer C compiler<sup>[1]</sup> used escape sequences to represent characters not always available on keywords of the day.

---

65) The semantics of these characters were discussed in 5.2.2.

### Commentary

This discussion describes the effect of writing these characters to a display device.

character  
display se-  
mantics

digraphs

footnote  
65

character  
display se-  
mantics

880 If any other character follows a backslash, the result is not a token and a diagnostic is required.

### Commentary

Such a sequence of characters does not form a valid preprocessing token (e.g., `'\D'` contains the preprocessing tokens `{'}`, `{\}`, `{D}`, and `{'}`, not the preprocessing token `{'\D'}`). When a preprocessing token contains a single-quote, the behavior is undefined.

character  
' or " matches

character  
' or " matches

It is possible that an occurrence of such a character sequence will cause a violation of syntax to occur, which will in turn require a diagnostic to be generated. However, implementations are not required to issue a diagnostic just because a footnote (which is not non-normative) says one is required.

### C90

*If any other escape sequence is encountered, the behavior is undefined.*

### C++

There is no equivalent sentence in the C++ Standard. However, this footnote is intended to explicitly spell out what the C syntax specifies. The C++ syntax specification is identical, but the implications have not been explicitly called out.

### Common Implementations

Most C90 translators do not issue a diagnostic for this violation of syntax. They treat all characters between matching single-quotes as forming an acceptable character constant.

### Coding Guidelines

There is existing practice that treats a backslash followed by another character as being just that character (except for the special cases described previously). The amount of source that contains this usage is unknown. Removing the unnecessary backslash is straight-forward for human-written code, but could be almost impossible for automatically generated code (access to the source of the generator is not always easy to obtain). Customer demand will ensure that translators continue to have an option to support existing practice.

881 See “future language directions” (6.11.4).

### Constraints

882 The value of an octal or hexadecimal escape sequence shall be in the range of representable values for the type **unsigned char** for an integer character constant, or the unsigned type corresponding to **wchar\_t** for a wide character constant.

escape sequence  
value within range

### Commentary

A character can be represented in an object of type **char**, and all members of the basic source character set are required to be represented using positive values. This constraint can be thought of as corresponding to the general constraint given for constants being representable within their type. (Although integer character constants have type **int** they are generally treated as having a character type.)

char  
hold any member of execution character set  
basic character set  
positive if stored in char object  
constant  
representable in its type

### C++

*The value of a character literal is implementation-defined if it falls outside of the implementation-defined range defined for char (for ordinary literals) or wchar\_t (for wide literals).*

2.13.2p4

The wording in the C++ Standard applies to the entire character literal, not to just a single character within it (the C case). In practice this makes no difference because C++ does not provide the option available to C implementations of allowing more than one character in an integer character constant.

The range of values that can be represented in the type **char** may be a subset of those representable in the type **unsigned char**. In some cases defined behavior in C becomes implementation-defined behavior in C++.

```
1 char *p = "\0x80"; /* does not affect the conformance status of the program */
2                 // if CHAR_MAX is 127, behavior is implementation-defined
```

In C a value outside of the representable range causes a diagnostic to be issued. The C++ behavior is implementation-defined in this case. Source developed using a C++ translator may need to be modified before it is acceptable to a C translator.

### Common Implementations

Some implementations perform a modulo operation on the value of the escape sequence to ensure it is within the representable range of the type **unsigned char**.

### Example

```
1 unsigned char uc_1 = '\xffff';
2 signed char sc_1 = '\xff';
```

## Semantics

An integer character constant has type **int**.

883

### Commentary

A character constant is essentially a way of representing the value of a character in the execution character set in a notation that is translator independent. C does not consider a character constant to be a special case of a string literal of length one (as some other languages do).

### C++

2.13.2p1 *An ordinary character literal that contains a single c-char has type **char**, . . .*

The only visible effect of this difference in type, from the C point of view, is the value returned by `sizeof`. In the C++ case the value is always 1, while in C the value is the same as `sizeof(int)`, which could have the value 1 (for some DSP chips), but for most implementations is greater than 1.

2.13.2p1 *A multicharacter literal has type **int** and implementation-defined value.*

The behavior in this case is identical to C.

### Other Languages

In most other languages a character constant has a character type.

### Coding Guidelines

A common developer misconception is that integer character constants have type **char** rather than **int**. Apart from the C++ compatibility issue and character constants appearing as the immediate operand of the `sizeof` operator, it is unlikely that there will be any cascading consequences following from this misconception.

The value of an integer character constant containing a single character that maps to a single-byte execution character is the numerical value of the representation of the mapped character interpreted as an integer. 884

## Commentary

This mapping can occur in one of two contexts— translation phase 1, the results of which are used during preprocessing, and during translation phase 5. There is no requirement that the two mappings be the same.

translation phase  
1  
translation phase  
5  
footnote  
141

```

1  #include <stdio.h>
2
3  void f(void)
4  {
5  if ('a' == 97)
6      printf("'a' == 97 in translation phase 5\n");
7
8  #if 'a' == 97
9  printf("'a' == 97 in translation phase 1\n");
10 #endif
11 }
```

The mapping does not apply to any character constants that are denoted using octal or hexadecimal escape sequences (a fact confirmed by the response to DR #017, question 4 and 5).

## Coding Guidelines

While translators treat character constants as numeric values internally, should developers be able to treat such constants as numeric values rather than symbolic quantities? The value of a character constant, which is not specified using an escape sequence, is representation information that depends on the implementation. (Although some of their properties are known, their range is specified and the digit characters have a contiguous encoding.) The guideline recommendation dealing with the use of representation information is applicable here.

882 escape sequence  
value within range  
digit characters contiguous  
?? representation information using

In some application domains a single representation is used for an implementations execution character set. Many character sets order the bit representations of their characters so that related characters have very similar bit patterns. For instance, corresponding upper- and lowercase letters are differentiated by a single bit in the Ascii character set, information that developers sometimes make use of to convert between upper- and lowercase (even though there are library functions available to perform the conversion). The following discussion looks at the issues involved in making use of character set representation details.

On seeing an arithmetic, bitwise, or relational operation involving a character constant, a reader needs to perform additional cognitive work that is not usually needed for other kinds of operations on this kind of operand, including:

- Performing a cognitive task switch. Character constants are usually thought of in symbolic rather than numeric terms. This is the rationale behind the lack of a guideline recommending that character constants be denoted, in the visible source by names, because of the strong semantic associations to readers of the source, created from experience in reading text.
- Recalling knowledge of the expected execution-time representation of character values. This is needed to deduce the intended consequences of the operation; for instance, using a bit-or operator to convert uppercase to lowercase.
- Deciding whether the result should continue to be treated as a symbolic quantity, or whether further operations are numeric in nature.

cognitive switch

866 character constant syntax

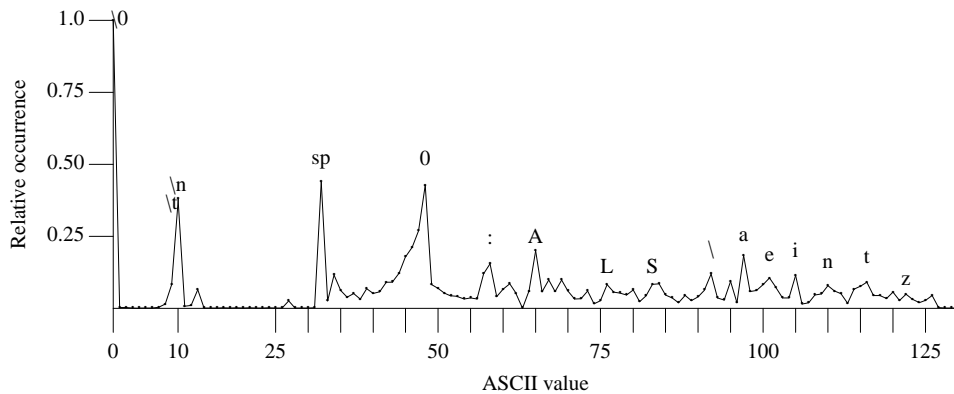
An integer character constant appearing as the operand of a bitwise, arithmetic, or relational operator is making use of representation information and is covered by a guideline recommendation.

?? representation information using

## Example

```

1  #define mk_lower_letter(position) ('a' + (position))
2
```



**Figure 884.1:** Relative frequency of occurrence of characters in an integer *character-constant* (as a fraction of the most common character, the null character). Based on the visible form of the `.c` files.

```

3 void f(char p1, int p2)
4 {
5   if ('a' < 'b')
6     ;
7   if ('a' < 97)
8     ;
9
10  p1++;      /* Was p1 last assigned a decimal constant or an integer character constant? */
11  p1 = 'a' + 3; /* Two different representation of an int being added. */
12  p1 = 'a' * 2; /* What does multiplying 'a' by 2 mean? */
13  p1 = 'a' + p2; /* The value of p2 is probably not known at translation time. */
14 }

```

**Table 884.1:** Occurrence of a *character-constant* appearing as one of the operands of various kinds of binary operators (as a percentage of all such constants; includes escape sequences). Based on the visible form of the `.c` files. See Table 866.3 for more detailed information.

Operator	%
Arithmetic operators	4.5
Bit operators	0.5
Equality operators	31.3
Relational operators	4.1

character  
constant  
more than one  
character

The value of an integer character constant containing more than one character (e.g., `'ab'`), or containing a character or escape sequence that does not map to a single-byte execution character, is implementation-defined.

885

### Commentary

Why does the C Standard allow for the possibility of more than one character in an integer character constant? There is some historical practice in this area. There is also a long-standing practice (particularly in Fortran) of packing several characters into an object having a larger integer type.

### C90

The value of an integer character constant containing more than one character, or containing a character or escape sequence not represented in the basic execution character set, is implementation-defined.

## C++

The C++ Standard does not include any statement covering escape sequences that are not represented in the execution character set. The other C requirements are covered by words (2.13.2p1) having the same meaning.

889 wide character escape sequence implementation-defined

## Other Languages

A character constant, in the lexical sense, that can contain more than one character is unique to C.

## Common Implementations

Given that an integer character constant has type **int**, most implementations support as many characters as there are bytes in that type, within character constants. The ordering of the characters within the representation of the type **int** varies between implementations. Possibilities, when the type **int** occupies four bytes, include (where underscores indicate padding bytes):

```
'ab'  => __ab, ab__, ba__
'abcd' => abcd, badc, dcba
```

The implementation-defined behavior of most implementations, for an escape sequence denoting a value outside of the range supported by the type **char** (but within the supported range of the type **unsigned char**), is to treat the most significant bit of the value as a sign bit. For instance, if the type **char** occupies eight bits, the value of the escape sequence `'\xFF'` is -1— if it is treated as a signed type. If it is treated as an unsigned type, the same escape sequence would have value 255. The sequence of casts `(int)(char)(unsigned char)0xFF` is one way of thinking about the conversion process.

## Coding Guidelines

The representation, in storage during program execution, of an integer character constant containing more than one character depends on implementation-defined behavior; it also defeats the rationale of using a character constant (i.e., its character'ness). The guideline recommendation dealing with the use of representation information is applicable.

?? representation information using

**Table 885.1:** Number of *character-constants* containing a given number of characters. Based on the visible form of the `.c` files.

Number of Characters	Occurrences	Number of Characters	Occurrences
0	27	4	21
1	50,590	5	4
2	0	6	4
3	8	7	0

886 If an integer character constant contains a single character or escape sequence, its value is the one that results when an object with type **char** whose value is that of the single character or escape sequence is converted to type **int**.

character constant single character value

## Commentary

A member of the basic execution character set can be represented in an object of type **char**, and all such members are represented using positive values. Under this model an escape sequence can be thought of as representing a sequence of bits. The value of this bit representation must be representable in the type **unsigned char**, but the actual value used is the bit representation treated as having a type **char**. The type **char** supports the same range of values as either the signed or unsigned character types, and the value of an escape sequence must be representable in the type **unsigned char**. If the type **char** is treated as being signed, any escape sequences whose value is greater than `SCHAR_MAX` results in implementation-defined behavior.

char hold any member of execution character set basic character set positive if stored in char object 882 escape sequence value within range char range, representation and behavior 882 escape sequence value within range

**C++**

The requirement contained in this sentence is not applicable to C++ because this language gives character literals the type **char**. There is no implied conversion to **int** in C++.

**Other Languages**

In most other languages character constants have type **char** and these issues do not apply.

**Coding Guidelines**

It is possible that a character constant may denote a value that is not representable in some implementations **char** type. An implementation's inability to support the desired behavior for these values is an issue that may need to be taken into account when selecting which vendors' translator to use. A guideline recommending against creating such a character constant serves no practical purpose (it is assumed that such a value was denoted using a character constant for a purpose).

---

A wide character constant has type **wchar\_t**, an integer type defined in the `<stddef.h>` header.

887

**Commentary**

It is the responsibility of the implementation to ensure that the definition of **wchar\_t** contained in the `<stddef.h>` header is compatible with the internal type used by the translator to assign a type to wide character constants. A developer-defined typedef whose name is **wchar\_t** does not have any affect on the type of a wide character constant, as per the behavior for **sizeof** and **size\_t**.

**C++**

2.13.2p2 *A wide-character literal has type **wchar\_t**.<sup>23)</sup>*

In C++ **wchar\_t** is one of the basic types (it is also a keyword). There is no need to define it in the `<stddef.h>` header.

3.9.1p5 *Type **wchar\_t** shall have the same size, signedness, and alignment requirements (3.9) as one of the other integral types, called its underlying type.*

Although C++ includes the C library by reference, the `<stddef.h>` header, in a C++ implementation, cannot contain a definition of the type **wchar\_t**, because **wchar\_t** is a keyword in C++. It is thus possible to use the type **wchar\_t** in C++ without including the `<stddef.h>` header.

**Other Languages**

In Java all character constants are capable of representing the full Unicode character set; there is no distinction between ordinary characters and wide characters.

**Common Implementations**

The type **wchar\_t** usually has a character type on implementations that only support the basic execution character set.

**Coding Guidelines**

A program that contains wide characters is also likely to use the type **wchar\_t** and the coding guideline issues are discussed under that type.

---

The value of a wide character constant containing a single multibyte character that maps to a member of the extended execution character set is the wide character corresponding to that multibyte character, as defined by the **mbtowc** function, with an implementation-defined current locale.

888

wide character constant type of

sizeof result type

multibyte character mapped by mbtowc

## Commentary

This requirement applies to members of the extended character set, not to members of the basic source character set. extended character set

From the practical point of view, translators need at least some of the members of the basic source character set to always map to the same values. For instance, the format specifiers used in the `scanf` family of functions treat white space as a special directive. Multibyte characters in the format string are converted to widechars in parsing the format, but there is no `isspace` library function to test for these characters. The only method is to check whether a character value is within the range of the basic execution character set and use the `isspace` library function. The translator behaves as-if the program fragment:

```
1  save_locale();
2  setlocale("LC_ALL", current_locale);
3  value = mbtowc(wide_character_constant);
4  restore_locale();
```

were executed in translation phase 7, where `curr_locale` is the implementation-defined current locale. The call to the `mbtowc` function cannot use the C locale by default because that locale does not define any extended characters. A related issue is discussed elsewhere. extended characters Lx == x

## C++

*The value of a wide-character literal containing a single `c-char` has value equal to the numerical value of the encoding of the `c-char` in the execution wide-character set.* 2.13.2p2

The C++ Standard includes the `mbtowc` function by including the C90 library by reference. However, it does not contain any requirement on the values of wide character literals corresponding to the definitions given for the `mbtowc` function (and its associated locale).

There is no requirement for C++ implementations to use a wide character mapping corresponding to that used by the `mbtowc` library function. However, it is likely that implementations of the two languages, in a given environment, will share the same library.

## Coding Guidelines

Accepting the implementation-defined behavior inherent in using wide string literals is part of the decision process that needs to be gone through when using any extended character set. One of the behaviors that developers need to check is whether the implementation-defined locale used by the translator is the same as the locale used by the program during execution.

889 The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set, is implementation-defined. wide character escape sequence implementation-defined

## Commentary

Support for wide character constants containing more than one multibyte character is consistent with such support for integer character constants. This specification requires the multibyte character to be a member of the extended execution character set. The equivalent wording for integer character constants requires that the value map to single-byte execution character. 885 character constant more than one character

## C++

The C++ Standard (2.13.2p2) does not include any statement covering escape sequences that are not represented in the execution character set.

**Example**

```

1  #include <wchar.h>
2
3  wchar_t w_1 = L'\xff',
4          w_2 = L'ab';

```

---

EXAMPLE 1 The construction `'\0'` is commonly used to represent the null character.

890

**Commentary**

null character

There are many ways of representing the null character. The construction `'\0'` is the shortest *character-constant*. This character sequence is also used within string literals to denote the null character.

---

EXAMPLE 2 Consider implementations that use two's-complement representation for integers and eight bits for objects that have type `char`. In an implementation in which type `char` has the same range of values as `signed char`, the integer character constant `'\xFF'` has the value `-1`; if type `char` has the same range of values as `unsigned char`, the character constant `'\xFF'` has the value `+255`.

891

**Commentary**

This difference of behavior, involving hexadecimal escape sequences, is often a novice developer's first encounter with the changeable representation of the type `char`.

---

EXAMPLE 3 Even if eight bits are used for objects that have type `char`, the construction `'\x123'` specifies an integer character constant containing only one character, since a hexadecimal escape sequence is terminated only by a non-hexadecimal character. To specify an integer character constant containing the two characters whose values are `'\x12'` and `'3'`, the construction `'\0223'` may be used, since an octal escape sequence is terminated after three octal digits. (The value of this two-character integer character constant is implementation-defined.)

892

**Commentary**

An example of what is specified in the syntax.

---

EXAMPLE 4 Even if 12 or more bits are used for objects that have type `wchar_t`, the construction `L'\1234'` specifies the implementation-defined value that results from the combination of the values `0123` and `'4'`.

893

**Commentary**

What is meant by “combination of the values” is not defined by the standard. The construction `L'\1234'` might be treated as equivalent to the wide character constant `L'S4'` (`S` having the Ascii value `0123`), or some alternative implementation-defined behavior may occur.

---

**Forward references:** common definitions `<stddef.h>` (7.17), the `mbtowc` function (7.20.7.2).

894

wide charac-  
ter escape  
sequence  
implementation-  
defined

## References

1. Perkin-Elmer. *C PROGRAMMING Manual*. Perkin-Elmer Cor-

poration, Oceanport, New Jersey 07757, 1984.