6.4.4 Constants

Syntax

1 *constant:*
integer-constant
floating-constant
enumeration-constant
character-constant

Constraints

2 Each constant shall have a type and the value of a constant shall be in the range of representable values for its type.

Semantics

3 Each constant has a type, determined by its form and value, as detailed later.

6.4.4.1 Integer constants

Syntax

1 *integer-constant:*
decimal-constant *integer-suffix*$_{\text{opt}}$
octal-constant *integer-suffix*$_{\text{opt}}$
hexadecimal-constant *integer-suffix*$_{\text{opt}}$

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*$_{\text{opt}}$
unsigned-suffix *long-long-suffix*
long-suffix *unsigned-suffix*$_{\text{opt}}$
long-long-suffix *unsigned-suffix*$_{\text{opt}}$

unsigned-suffix: one of
u U

long-suffix: one of
l L

long-long-suffix: one of
ll LL

Description

2 An integer constant begins with a digit, but has no period or exponent part. It may have a prefix that specifies its base and a suffix that specifies its type.

3 A decimal constant begins with a nonzero digit and consists of a sequence of decimal digits. An octal constant consists of the prefix **0** optionally followed by a sequence of the digits **0** through **7** only. 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.

Semantics

4 The value of a decimal constant is computed base 10; that of an octal constant, base 8; that of a hexadecimal constant, base 16. The lexically first digit is the most significant.

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

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**

6 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. If all of the types in the list for the constant are signed, the extended integer type shall be signed. If all of the types in the list for the constant are unsigned, the extended integer type shall be unsigned. If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned. 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.

6.4.4.2 Floating constants

Syntax

1 *floating-constant:*
decimal-floating-constant
hexadecimal-floating-constant

decimal-floating-constant:
fractional-constant *exponent-part*$_{\text{opt}}$ *floating-suffix*$_{\text{opt}}$
digit-sequence *exponent-part* *floating-suffix*$_{\text{opt}}$

hexadecimal-floating-constant:
hexadecimal-prefix *hexadecimal-fractional-constant*
binary-exponent-part *floating-suffix*$_{\text{opt}}$
hexadecimal-prefix *hexadecimal-digit-sequence*
binary-exponent-part *floating-suffix*$_{\text{opt}}$

fractional-constant:
digit-sequence$_{\text{opt}}$ **.** *digit-sequence*
digit-sequence **.**

exponent-part:
e *sign*$_{\text{opt}}$ *digit-sequence*
E *sign*$_{\text{opt}}$ *digit-sequence*

sign: one of
+ -

digit-sequence:
digit
digit-sequence *digit*

hexadecimal-fractional-constant:
hexadecimal-digit-sequence$_{\text{opt}}$ **.**
hexadecimal-digit-sequence
hexadecimal-digit-sequence **.**

binary-exponent-part:
p *sign*$_{\text{opt}}$ *digit-sequence*
P *sign*$_{\text{opt}}$ *digit-sequence*

hexadecimal-digit-sequence:
hexadecimal-digit
hexadecimal-digit-sequence *hexadecimal-digit*

floating-suffix: one of
f l F L

Description

2 A floating constant has a *significand part* that may be followed by an *exponent part* and a suffix that specifies its type. The components of the significand part may include a digit sequence representing the whole-number part, followed by a period (**.**), followed by a digit sequence representing the fraction part. The components of the exponent part are an **e**, **E**, **p**, or **P** followed by an exponent consisting of an optionally signed digit sequence. Either the whole-number part or the fraction part has to be present; for decimal floating constants, either the period or the exponent part has to be present.

Semantics

3 The significand part is interpreted as a (decimal or hexadecimal) rational number; the digit sequence in the exponent part is interpreted as a decimal integer. For decimal floating constants, the exponent indicates the power of 10 by which the significand part is to be scaled. For hexadecimal floating constants, the exponent indicates the power of 2 by which the significand part is to be scaled. For decimal floating constants, and also for hexadecimal floating constants when **FLT_RADIX** is not a power of 2, the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner. For hexadecimal floating constants when **FLT_RADIX** is a power of 2, the result is correctly rounded.

4 An unsuffixed floating constant has type **double**. If suffixed by the letter **f** or **F**, it has type **float**. If suffixed by the letter **l** or **L**, it has type **long double**.

5 Floating constants are converted to internal format as if at translation-time. The conversion of a floating constant shall not raise an exceptional condition or a floating-point exception at execution time. All floating constants of the same source form[76)] shall convert to the same internal format with the same value.

Recommended practice

6 The implementation should produce a diagnostic message if a hexadecimal constant cannot be represented exactly in its evaluation format; the implementation should then proceed with the translation of the program.

7 The translation-time conversion of floating constants should match the execution-time conversion of character strings by library functions, such as **strtod**, given matching inputs suitable for both conversions, the same result format, and default execution-time rounding.[77)]

6.4.4.3 Enumeration constants

Syntax

1 *enumeration-constant:*
identifier

Semantics

2 An identifier declared as an enumeration constant has type **int**.

Forward references: enumeration specifiers (6.7.2.2).

6.4.4.4 Character constants

Syntax

1 *character-constant:*
' *c-char-sequence* '
L' *c-char-sequence* '
u' *c-char-sequence* '
U' *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

[76)] `1.23`, `1.230`, `123e-2`, `123e-02`, and `1.23L` are all different source forms and thus need not convert to the same internal format and value.

[77)] The specification for the library functions recommends more accurate conversion than required for floating constants (see 7.22.1.3).

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

Description

2 An integer character constant is a sequence of one or more multibyte characters enclosed in single-quotes, as in `'x'`. A wide character constant is the same, except prefixed by the letter **`L`**, **`u`**, or **`U`**. With a few exceptions detailed later, the elements of the sequence are any members of the source character set; they are mapped in an implementation-defined manner to members of the execution character set.

3 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:

single quote '	`\'`
double quote "	`\"`
question mark ?	`\?`
backslash \	`\\`
octal character	`\`*octal digits*
hexadecimal character	`\x` *hexadecimal digits*

4 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 `\\`.

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

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

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

8 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`.[78)]

[78)] The semantics of these characters were discussed in 5.2.2. If any other character follows a backslash, the result is not a token and a diagnostic is required. See "future language directions" (6.11.4).

Constraints

9 The value of an octal or hexadecimal escape sequence shall be in the range of representable values for the corresponding type:

Prefix	Corresponding Type
none	**unsigned char**
L	the unsigned type corresponding to **wchar_t**
u	**char16_t**
U	**char32_t**

Semantics

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

11 A wide character constant prefixed by the letter L has type **wchar_t**, an integer type defined in the <stddef.h> header; a wide character constant prefixed by the letter u or U has type **char16_t** or **char32_t**, respectively, unsigned integer types defined in the <uchar.h> header. The value of a wide character constant containing a single multibyte character that maps to a single member of the extended execution character set is the wide character corresponding to that multibyte character, as defined by the **mbtowc**, **mbrtoc16**, or **mbrtoc32** function as appropriate for its type, with an implementation-defined current locale. The value of a wide character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a multibyte character or escape sequence not represented in the extended execution character set, is implementation-defined.

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

13 **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 $+255$.

14 **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.)

15 **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'.

Forward references: common definitions <stddef.h> (7.19), the **mbtowc** function (7.22.7.2), Unicode utilities <uchar.h> (7.28).

6.4.5 String literals

Syntax

1 *string-literal:*

encoding-prefix$_{\text{opt}}$ " *s-char-sequence*$_{\text{opt}}$ "

encoding-prefix:
u8
u
U
L

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
any member of the source character set except
the double-quote ", backslash \, or new-line character
escape-sequence

Constraints

2 A sequence of adjacent string literal tokens shall not include both a wide string literal and a UTF–8 string literal.

Description

3 A *character string literal* is a sequence of zero or more multibyte characters enclosed in double-quotes, as in `"xyz"`. A UTF–*8 string literal* is the same, except prefixed by **u8**. A *wide string literal* is the same, except prefixed by the letter **L**, **u**, or **U**.

4 The same considerations apply to each element of the sequence in a string literal as if it were in an integer character constant (for a character or UTF–8 string literal) or a wide character constant (for a wide string literal), except that the single-quote ' is representable either by itself or by the escape sequence `\'`, but the double-quote " shall be represented by the escape sequence `\"`.

Semantics

5 In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character and identically-prefixed string literal tokens are concatenated into a single multibyte character sequence. If any of the tokens has an encoding prefix, the resulting multibyte character sequence is treated as having the same prefix; otherwise, it is treated as a character string literal. Whether differently-prefixed wide string literal tokens can be concatenated and, if so, the treatment of the resulting multibyte character sequence are implementation-defined.

6 In translation phase 7, a byte or code of value zero is appended to each multibyte character sequence that results from a string literal or literals.[79)] The multibyte character sequence is then used to initialize an array of static storage duration and length just sufficient to contain the sequence. For character string literals, the array elements have type **char**, and are initialized with the individual bytes of the multibyte character sequence. For UTF–8 string literals, the array elements have type **char**, and are initialized with the characters of the multibyte character sequence, as encoded in UTF–8. For wide string literals prefixed by the letter **L**, the array elements have type **wchar_t** and are initialized with the sequence of wide characters corresponding to the multibyte character sequence, as defined by the **mbstowcs** function with an implementation-defined current locale. For wide string literals prefixed by the letter **u** or **U**, the array elements have type **char16_t** or **char32_t**, respectively, and are initialized with the sequence of wide characters corresponding to the multibyte character sequence, as defined by successive calls to the **mbrtoc16**, or **mbrtoc32** function as appropriate for its type, with an implementation-defined current locale. The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set is implementation-defined.

7 It is unspecified whether these arrays are distinct provided their elements have the appropriate values. If the program attempts to modify such an array, the behavior is undefined.

[79)] A string literal need not be a string (see 7.1.1), because a null character may be embedded in it by a `\0` escape sequence.

8 **EXAMPLE 1** This pair of adjacent character string literals

```
"\x12" "3"
```

produces a single character string literal containing the two characters whose values are `'\x12'` and `'3'`, because escape sequences are converted into single members of the execution character set just prior to adjacent string literal concatenation.

9 **EXAMPLE 2** Each of the sequences of adjacent string literal tokens

```
"a" "b" L"c"
"a" L"b" "c"
L"a" "b" L"c"
L"a" L"b" L"c"
```

is equivalent to the string literal

```
L"abc"
```

Likewise, each of the sequences

```
"a" "b" u"c"
"a" u"b" "c"
u"a" "b" u"c"
u"a" u"b" u"c"
```

is equivalent to

```
u"abc"
```

Forward references: common definitions <stddef.h> (7.19), the **mbstowcs** function (7.22.8.1), Unicode utilities <uchar.h> (7.28).

6.4.6 Punctuators

Syntax

1 *punctuator:* one of

```
[ ] ( ) { } . ->
++ -- & * + - ~ !
/ % << >> < > <= >= == != ^ | && ||
? : ; ...
= *= /= %= += -= <<= >>= &= ^= |=
, # ##
<: :> <% %> %: %:%:
```

Semantics

2 A punctuator is a symbol that has independent syntactic and semantic significance. Depending on context, it may specify an operation to be performed (which in turn may yield a value or a function designator, produce a side effect, or some combination thereof) in which case it is known as an *operator* (other forms of operator also exist in some contexts). An *operand* is an entity on which an operator acts.

3 In all aspects of the language, the six tokens[80)]

```
<: :> <% %> %: %:%:
```

behave, respectively, the same as the six tokens

```
[ ] { } # ##
```

except for their spelling.[81)]

[80)] These tokens are sometimes called "digraphs".

[81)] Thus `[` and `<:` behave differently when "stringized" (see 6.10.3.2), but can otherwise be freely interchanged.