Floating-point literal

Floating-point literal defines a compile-time constant whose value is specified in the source file.

Syntax

digit-sequence decimal-exponent suffix(optional)	(1)	
digit-sequence **.** *decimal-exponent*(optional) *suffix*(optional)	(2)	
digit-sequence(optional) **.** *digit-sequence decimal-exponent*(optional) *suffix*(optional)	(3)	
0x | **0X** *hex-digit-sequence hex-exponent suffix*(optional)	(4)	(since C++17)
0x | **0X** *hex-digit-sequence* **.** *hex-exponent suffix*(optional)	(5)	(since C++17)
0x | **0X** *hex-digit-sequence*(optional) **.** *hex-digit-sequence hex-exponent suffix*(optional)	(6)	(since C++17)

1) *digit-sequence* representing a whole number without a decimal separator, in this case the exponent is not optional: `1e10`, `1e-5L`
2) *digit-sequence* representing a whole number with a decimal separator, in this case the exponent is optional: `1.`, `1.e-2`
3) *digit-sequence* representing a fractional number. The exponent is optional: `3.14`, `.1f`, `0.1e-1L`
4) Hexadecimal *digit-sequence* representing a whole number without a radix separator. The exponent is never optional for hexadecimal floating-point literals: `0x1ffp10`, `0X0p-1`
5) Hexadecimal *digit-sequence* representing a whole number with a radix separator. The exponent is never optional for hexadecimal floating-point literals: `0x1.p0`, `0xf.p-1`
6) Hexadecimal *digit-sequence* representing a fractional number with a radix separator. The exponent is never optional for hexadecimal floating-point literals: `0x0.123p-1`, `0xa.bp10l`

decimal-exponent has the form

e | **E** *exponent-sign*(optional) *digit-sequence*

hex-exponent has the form

p | **P** *exponent-sign*(optional) *digit-sequence*	(since C++17)

exponent-sign, if present, is either **+** or **-**

suffix, if present, is one of **f**, **l**, **F**, **L**, **f16**, **f32**, **f64**, **f128**, **bf16**, **F16**, **F32**, **F64**, **F128**, **BF16** (since C++23). The suffix determines the type of the floating-point literal:

- (no suffix) defines `double`
- **f** **F** defines `float`
- **l** **L** defines `long double`

- **f16** **F16** defines `std::float16_t`
- **f32** **F32** defines `std::float32_t`
- **f64** **F64** defines `std::float64_t` (since C++23)
- **f128** **F128** defines `std::float128_t`
- **bf16** **BF16** defines `std::bfloat16_t`

Optional single quotes (**'**) may be inserted between the digits as a separator; they are ignored during compilation. (since C++14)

Explanation

Decimal scientific notation is used, meaning that the value of the floating-point literal is the significand multiplied by the number 10 raised to the power of *decimal-exponent*. E.g. the mathematical meaning of `123e4` is 123×10^4.

If the floating literal begins with the character sequence `0x` or `0X`, the floating literal is a *hexadecimal floating literal*. Otherwise, it is a *decimal floating literal*.

For a *hexadecimal floating literal*, the significand is interpreted as a hexadecimal rational number, and the *digit-sequence* of the exponent is interpreted as the (decimal) integer power of 2 by which the significand has to be scaled. (since C++17)

```
double d = 0x1.4p3;  // hex fraction 1.4 (decimal 1.25) scaled by 2^3, that is 10.0
```

Notes

The hexadecimal floating-point literals were not part of C++ until C++17, although they can be parsed and printed by the I/O functions since C++11: both C++ I/O streams when `std::hexfloat` is enabled and the C I/O streams: `std::printf`, `std::scanf`, etc. See `std::strtof` for the format description.

Feature-test macro	Value	Std	Comment
__cpp_hex_float	201603L	(C++17)	Hexadecimal floating literals

Example

Run this code

```
#include <iostream>
#include <iomanip>
#include <limits>
#include <typeinfo>

#define OUT(x) '\n' << std::setw(16) << #x << x

int main()
{
    std::cout
        << "Literal" "\t" "Printed value" << std::left
        << OUT( 58.             ) // double
        << OUT( 4e2             ) // double
        << OUT( 123.456e-67     ) // double
        << OUT( 123.456e-67f    ) // float, truncated to zero
        << OUT( .1E4f           ) // float
        << OUT( 0x10.1p0        ) // double
        << OUT( 0x1p5           ) // double
        << OUT( 0x1e5           ) // integer literal, not floating-point
        << OUT( 3.14'15'92      ) // double, single quotes ignored (C++14)
        << OUT( 1.18e-4932l     ) // long double
        << std::setprecision(39)
        << OUT( 3.4028234e38f   ) // float
        << OUT( 3.4028234e38    ) // double
        << OUT( 3.4028234e38l   ) // long double
        << '\n';

    static_assert(3.4028234e38f == std::numeric_limits<float>::max());

    static_assert(3.4028234e38f ==  // ends with 4
                  3.4028235e38f);   // ends with 5

    static_assert(3.4028234e38 !=   // ends with 4
                  3.4028235e38);    // ends with 5

    // Both floating-point constants below are 3.4028234e38
    static_assert(3.4028234e38f !=  // a float (then promoted to double)
                  3.4028234e38);    // a double
}
```

Possible output:

```
Literal         Printed value
58.             58
4e2             400
123.456e-67     1.23456e-65
123.456e-67f    0
.1E4f           1000
0x10.1p0        16.0625
```

```
0x1p5           32
0x1e5           485
3.14'15'92      3.14159
1.18e-4932l     1.18e-4932
3.4028234e38f   340282346638528859811704183484516925440
3.4028234e38    340282339999999992395853996843190976512
3.4028234e38l   340282339999999999959125552115262423 04
```

See also

user-defined literals(C++11)	literals with user-defined suffix

C documentation for **Floating constant**