

Binary Literals: The 0b Prefix

The 0b (or 0B) prefix, modeled after 0x, enables **integer literals** to be expressed in base 2.

Description

A binary **literal** is an integral value represented in code in a binary numeral system. A binary literal consists of a 0b or 0B prefix followed by a nonempty sequence of binary digits, namely, 0 and 1¹:

```
int i = 0b11110000; // equivalent to 240, 0360, or 0xF0
int j = 0B11110000; // same value as above
```

The first digit after the 0b prefix is the most significant one:

```
static_assert(0b0 == 0, ""); // 0*2^0
static_assert(0b1 == 1, ""); // 1*2^0
static_assert(0b10 == 2, ""); // 1*2^1 + 0*2^0
static_assert(0b11 == 3, ""); // 1*2^1 + 1*2^0
static_assert(0b100 == 4, ""); // 1*2^2 + 0*2^1 + 0*2^0
static_assert(0b101 == 5, ""); // 1*2^2 + 0*2^1 + 1*2^0
// ...
static_assert(0b11010 == 26, ""); // 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0
```

Leading zeros — as with octal and hexadecimal (but not decimal) **literals** — are ignored but can be added for readability:

```
static_assert(0b00000000 == 0, "");
static_assert(0b00000001 == 1, "");
static_assert(0b00000010 == 2, "");
static_assert(0b00000100 == 4, "");
static_assert(0b00001000 == 8, "");
static_assert(0b10000000 == 128, "");
```

The type of a binary **literal**, just as with a decimal, octal, or hexadecimal **literal**, is determined by the compiler using a combination of the **literal’s** suffix, if any, and its **value**. For each **literal**, a sequence of types — initially **int**, **unsigned int**, **long**, **unsigned long**, **long long**, **unsigned long long** — is determined by the compiler. For unsigned **literals**, i.e., those having a u or U suffix, all *signed* types (e.g., **int**, **long long**) from the initial sequence above are removed by the compiler. For decimal (but not nondecimal) **literals** having no u or U suffix, all the *unsigned* types (e.g., **unsigned long**) are removed by the compiler. The presence of an l or L suffix will prompt the compiler to remove **int** and **unsigned int** from

¹Prior to being introduced in C++14, GCC supported binary **literals** — with the same syntax as the standard feature — as a nonconforming extension since version 4.3.0, released in March 2008; for more details, see <https://gcc.gnu.org/gcc-4.3/>.

Section 1.2 C++14

Binary Literals

the sequence, while an `ll` or `LL` will remove those types and `long` and `unsigned long` too. The earliest type in the resulting sequence that can represent the value of the literal will be used by the compiler. If no suitable type is found, an (implementation-defined) **extended integer type** of an appropriate size and signedness, such as `__int128` or `unsigned __int128`, may be used by the compiler; otherwise, the program is **ill formed**:

```
// example platform 1:
// (sizeof(int): 4; sizeof(long): 4; sizeof(long long): 8)
auto i32 = 0b0111...[ 24 1-bits]...1111; // i32 is int.
auto u32 = 0b1000...[ 24 0-bits]...0000; // u32 is unsigned int.
auto i64 = 0b0111...[ 56 1-bits]...1111; // i64 is long long.
auto u64 = 0b1000...[ 56 0-bits]...0000; // u64 is unsigned long long.
auto i128 = 0b0111...[120 1-bits]...1111; // Error, integer literal too large
auto u128 = 0b1000...[120 0-bits]...0000; // Error, integer literal too large

// example platform 2:
// (sizeof(int): 4; sizeof(long): 8; sizeof(long long): 16)
auto i32 = 0b0111...[ 24 1-bits]...1111; // i32 is int.
auto u32 = 0b1000...[ 24 0-bits]...0000; // u32 is unsigned int.
auto i64 = 0b0111...[ 56 1-bits]...1111; // i64 is long.
auto u64 = 0b1000...[ 56 0-bits]...0000; // u64 is unsigned long.
auto i128 = 0b0111...[120 1-bits]...1111; // i128 is long long.
auto u128 = 0b1000...[120 0-bits]...0000; // u128 is unsigned long long.
```

Purely for convenience of exposition, we have employed the C++11 `auto` feature to conveniently capture the type implied by the literal itself; see Section 2.1. “`auto` Variables” on page 195. Separately, the precise initial type of a binary literal, like any other literal, can be controlled explicitly using the common integer-literal suffixes `{u, l, ul, ll, ull}` in either lower- or uppercase:

```
auto i  = 0b101;           // type: int;           value: 5
auto u  = 0b1010U;        // type: unsigned int;       value: 10
auto l  = 0b1111L;        // type: long;               value: 15
auto ul = 0b10100UL;      // type: unsigned long;      value: 20
auto ll = 0b11000LL;      // type: long long;          value: 24
auto ull = 0b110101ULL;   // type: unsigned long long; value: 53
```

Finally, note that affixing a minus sign to a binary literal (e.g., `-b1010`) — just like any other integer literal (e.g., `-10`, `-012`, or `-0xa`) — is parsed as a non-negative value first, after which a unary minus is applied:

```
static_assert(sizeof(int) == 4, ""); // true on virtually all machines today
static_assert(-0b1010 == -10, ""); // as if: 0 - 0b1010 == 0 - 10
static_assert( 0b0111...[ 24 1-bits]...1111 // signed
              != -0b0111...[ 24 1-bits]...1111, ""); // signed

static_assert( 0b1000...[ 24 0-bits]...0000 // unsigned
              != -0b1000...[ 24 0-bits]...0000, ""); // unsigned
```