Fundamental types

(See also type for type system overview and the list of type-related utilities that are provided by the C++ library)

Void type

`void` - type with an empty set of values. It is an incomplete type that cannot be completed (consequently, objects of type `void` are disallowed). There are no arrays of `void`, nor references to `void`. However, pointers to `void` and functions returning type `void` (*procedures* in other languages) are permitted.

`std::nullptr_t`

Defined in header <cstddef>

```
typedef decltype(nullptr) nullptr_t;    (since C++11)
```

`std::nullptr_t` is the type of the null pointer literal, `nullptr`. It is a distinct type that is not itself a pointer type or a pointer to member type. Its values are *null pointer constant* (see `NULL`), and may be implicitly converted to any pointer and pointer to member type.

`sizeof(std::nullptr_t)` is equal to `sizeof(void *)`.

Data models

The choices made by each implementation about the sizes of the fundamental types are collectively known as *data model*. Four data models found wide acceptance:

32 bit systems:

- **LP32** or **2/4/4** (int is 16-bit, long and pointer are 32-bit)
  - Win16 API
- **ILP32** or **4/4/4** (int, long, and pointer are 32-bit);
  - Win32 API
  - Unix and Unix-like systems (Linux, macOS)

64 bit systems:

- **LLP64** or **4/4/8** (int and long are 32-bit, pointer is 64-bit)
  - Win64 API
- **LP64** or **4/8/8** (int is 32-bit, long and pointer are 64-bit)
  - Unix and Unix-like systems (Linux, macOS)

Other models are very rare. For example, **ILP64** (**8/8/8**: int, long, and pointer are 64-bit) only appeared in some early 64-bit Unix systems (e.g. UNICOS on Cray).

Signed and unsigned integer types

`int` - basic integer type. The keyword `int` may be omitted if any of the modifiers listed below are used. If no length modifiers are present, it's guaranteed to have a width of at least 16 bits. However, on 32/64 bit systems it is almost exclusively guaranteed to have width of at least 32 bits (see below).

Modifiers

Modifies the basic integer type. Can be mixed in any order. Only one of each group can be present in type name.

Signedness

`signed` - target type will have signed representation (this is the default if omitted)
`unsigned` - target type will have unsigned representation

Size

`short` - target type will be optimized for space and will have width of at least 16 bits.
`long` - target type will have width of at least 32 bits.

`long long` - target type will have width of at least 64 bits. (since C++11)

Note: as with all type specifiers, any order is permitted: `unsigned long long int` and `long int unsigned long` name the same type.

Properties

The following table summarizes all available integer types and their properties in various common data models:

Type specifier	Equivalent type	Width in bits by data model				
		C++ standard	LP32	ILP32	LLP64	LP64
`short`	`short int`	at least **16**	**16**	**16**	**16**	**16**
`short int`						
`signed short`						
`signed short int`						
`unsigned short`	`unsigned short int`					
`unsigned short int`						
`int`	`int`	at least **16**	**16**	**32**	**32**	**32**
`signed`						
`signed int`						
`unsigned`	`unsigned int`					
`unsigned int`						
`long`	`long int`	at least **32**	**32**	**32**	**32**	**64**
`long int`						
`signed long`						
`signed long int`						
`unsigned long`	`unsigned long int`					
`unsigned long int`						
`long long`	`long long int` (C++11)	at least **64**	**64**	**64**	**64**	**64**
`long long int`						
`signed long long`						
`signed long long int`						
`unsigned long long`	`unsigned long long int` (C++11)					
`unsigned long long int`						

Note: integer arithmetic is defined differently for the signed and unsigned integer types. See arithmetic operators, in particular integer overflows.

`std::size_t` is the unsigned integer type of the result of the `sizeof` operator as well as the `sizeof...` operator and the `alignof` operator (since C++11).

See also Fixed width integer types. (since C++11)

Boolean type

`bool` - type, capable of holding one of the two values: `true` or `false`. The value of `sizeof(bool)` is implementation defined and might differ from 1.

Character types

`signed char` - type for signed character representation.
`unsigned char` - type for unsigned character representation. Also used to inspect object representations (raw memory).
`char` - type for character representation which can be most efficiently processed on the target system (has the same representation and alignment as either `signed char` or `unsigned char`, but is always a distinct type). Multibyte characters strings use this type to represent code units. For every value of type `unsigned char` in range [0, 255], converting the value to `char` and then back to `unsigned char` produces the original value. (since C++11)
The signedness of `char` depends on the compiler and the target platform: the defaults for ARM and PowerPC are typically unsigned, the defaults for x86 and x64 are typically signed.
`wchar_t` - type for wide character representation (see wide strings). Required to be large enough to represent any supported character code point (32 bits on systems that support Unicode. A notable exception is Windows, where

wchar_t is 16 bits and holds UTF-16 code units) It has the same size, signedness, and alignment as one of the integer types, but is a distinct type.

`char16_t` - type for UTF-16 character representation, required to be large enough to represent any UTF-16 code unit (16 bits). It has the same size, signedness, and alignment as `std::uint_least16_t`, but is a distinct type. (since C++11)

`char32_t` - type for UTF-32 character representation, required to be large enough to represent any UTF-32 code unit (32 bits). It has the same size, signedness, and alignment as `std::uint_least32_t`, but is a distinct type.

`char8_t` - type for UTF-8 character representation, required to be large enough to represent any UTF-8 code unit (8 bits). It has the same size, signedness, and alignment as `unsigned char` (and therefore, the same size and alignment as `char` and `signed char`), but is a distinct type. (since C++20)

Besides the minimal bit counts, the C++ Standard guarantees that

`1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`.

Note: this allows the extreme case in which bytes are sized 64 bits, all types (including `char`) are 64 bits wide, and `sizeof` returns 1 for every type.

Floating-point types

The following three types and their cv-qualified versions are collectively called floating-point types.

`float` - single precision floating point type. Matches IEEE-754 binary32 format if supported.
`double` - double precision floating point type. Matches IEEE-754 binary64 format if supported.
`long double` - extended precision floating point type. Matches IEEE-754 binary128 format if supported, otherwise matches IEEE-754 binary64-extended format if supported, otherwise matches some non-IEEE-754 extended floating-point format as long as its precision is better than binary64 and range is at least as good as binary64, otherwise matches IEEE-754 binary64 format.

- binary128 format is used by some HP-UX, SPARC, MIPS, ARM64, and z/OS implementations.
- The most well known IEEE-754 binary64-extended format is 80-bit x87 extended precision format. It is used by many x86 and x86-64 implementations (a notable exception is MSVC, which implements `long double` in the same format as `double`, i.e. binary64).

Properties

Floating-point types may support special values:

- *infinity* (positive and negative), see `INFINITY`
- the *negative zero*, `-0.0`. It compares equal to the positive zero, but is meaningful in some arithmetic operations, e.g. `1.0/0.0 == INFINITY`, but `1.0/-0.0 == -INFINITY`), and for some mathematical functions, e.g. `sqrt(std::complex)`
- *not-a-number* (NaN), which does not compare equal with anything (including itself). Multiple bit patterns represent NaNs, see `std::nan`, `NAN`. Note that C++ takes no special notice of signalling NaNs other than detecting their support by `std::numeric_limits::has_signaling_NaN`, and treats all NaNs as quiet.

Real floating-point numbers may be used with arithmetic operators + - / * and various mathematical functions from cmath. Both built-in operators and library functions may raise floating-point exceptions and set `errno` as described in `math_errhandling`

Floating-point expressions may have greater range and precision than indicated by their types, see `FLT_EVAL_METHOD`. Floating-point expressions may also be *contracted*, that is, calculated as if all intermediate values have infinite range and precision, see #pragma STDC FP_CONTRACT.

Some operations on floating-point numbers are affected by and modify the state of the floating-point environment (most notably, the rounding direction)

Implicit conversions are defined between real floating types and integer types.

See Limits of floating point types and `std::numeric_limits` for additional details, limits, and properties of the floating-point types.

Range of values

The following table provides a reference for the limits of common numeric representations.

Prior to C++20, the C++ Standard allowed any signed integer representation, and the minimum guaranteed range of N-bit signed integers was from $-(2^{N-1}-1)$ to $+2^{N-1}-1$ (e.g. **-127** to **127** for a signed 8-bit type), which corresponds to the limits of one's complement or sign-and-magnitude.

However, all C++ compilers use two's complement representation, and as of C++20, it is the only representation allowed by the standard, with the guaranteed range from -2^{N-1} to $+2^{N-1}-1$ (e.g. **-128** to **127** for a signed 8-bit type).

8-bit one's complement and sign-and-magnitude representations for `char` have been disallowed since C++11 (via CWG 1759), because a UTF-8 code unit of value 0x80 used in a UTF-8 string literal must be storable in a `char` element object.

Type	Size in bits	Format	Value range	
			Approximate	Exact
character	8	signed		**-128** to **127**
		unsigned		**0** to **255**
	16	UTF-16		**0** to **65535**
	32	UTF-32		**0** to **1114111** (**0x10ffff**)
integer	16	signed	**± 3.27 · 10^4**	**-32768** to **32767**
		unsigned	**0** to **6.55 · 10^4**	**0** to **65535**
	32	signed	**± 2.14 · 10^9**	**-2,147,483,648** to **2,147,483,647**
		unsigned	**0** to **4.29 · 10^9**	**0** to **4,294,967,295**
	64	signed	**± 9.22 · 10^{18}**	**-9,223,372,036,854,775,808** to **9,223,372,036,854,775,807**
		unsigned	**0** to **1.84 · 10^{19}**	**0** to **18,446,744,073,709,551,615**
binary floating point	32	IEEE-754	▪ min subnormal: **± 1.401,298,4 · 10^{-45}** ▪ min normal: **± 1.175,494,3 · 10^{-38}** ▪ max: **± 3.402,823,4 · 10^{38}**	▪ min subnormal: **±0x1p-149** ▪ min normal: **±0x1p-126** ▪ max: **±0x1.fffffep+127**
	64	IEEE-754	▪ min subnormal: **± 4.940,656,458,412 · 10^{-324}** ▪ min normal: **± 2.225,073,858,507,201,4 · 10^{-308}** ▪ max: **± 1.797,693,134,862,315,7 · 10^{308}**	▪ min subnormal: **±0x1p-1074** ▪ min normal: **±0x1p-1022** ▪ max: **±0x1.fffffffffffffp+1023**
	80[note 1]	x86	▪ min subnormal: **± 3.645,199,531,882,474,602,528 · 10^{-4951}** ▪ min normal: **± 3.362,103,143,112,093,506,263 · 10^{-4932}** ▪ max: **± 1.189,731,495,357,231,765,021 · 10^{4932}**	▪ min subnormal: **±0x1p-16446** ▪ min normal: **±0x1p-16382** ▪ max: **±0x1.fffffffffffffffep+16383**
	128	IEEE-754	▪ min subnormal: **± 6.475,175,119,438,025,110,924,438,958,227,646,552,5 · 10^{-4966}** ▪ min normal: **± 3.362,103,143,112,093,506,262,677,817,321,752,602,6 · 10^{-4932}** ▪ max: **± 1.189,731,495,357,231,765,085,759,326,628,007,016,2 · 10^{4932}**	▪ min subnormal: **±0x1p-16494** ▪ min normal: **±0x1p-16382** ▪ max: **±0x1.ffffffffffffffffffffffffffffp+16383**

1. ↑ The object representation usually occupies 96/128 bits on 32/64-bit platforms respectively.

Note: actual (as opposed to guaranteed minimal) limits on the values representable by these types are available in C numeric limits interface and `std::numeric_limits`.

Keywords

`void`, `bool`, `true`, `false`, `char`, `wchar_t`, `char8_t`, `char16_t`, `char32_t`, `int`, `short`, `long`, `signed`, `unsigned`, `float`, `double`

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
CWG 1759 (https://wg21.cmeerw.net/cwg/issue1759)	C++11	`char` is not guaranteed to be able to represent UTF-8 code unit 0x80	guaranteed

See also

- the C++ type system overview
- const-volatility (cv) specifiers and qualifiers
- storage duration specifiers

C documentation for **arithmetic types**