

# Modern C++ Programming

## 3. BASIC CONCEPTS I

TYPE SYSTEM, FUNDAMENTAL TYPES, AND OPERATORS

---

*Federico Busato*

2026-04-18

## 1 The C++ Type System

- Type Categories
- Type Properties ★

## 2 Fundamental Types Overview

- Arithmetic Types
- Suffix and Prefix
- Non-Standard Arithmetic Types
- void Type
- `nullptr`

## 3 Conversion Rules

## 4 auto Keyword

## 5 C++ Operators

- Operators Precedence
- Prefix/Postfix Increment/Decrement Semantic
- Assignment, Compound, and Comma Operators
- Spaceship Operator `<=>` ★
- Safe Comparison Operators ★

# The C++ Type System

---

# The C++ Type System

C++ is a **strongly typed** and **statically typed** language

*Every entity has a type and that type never changes*

Every variable, function, or expression has a **type** in order to be compiled. Users can introduce new types with `class` or `struct`

The **type** specifies:

- The *amount of memory* allocated for the variable (or expression result)
- The *kinds of values* that may be stored and how the compiler interprets the bit patterns in those values
- The *operations* that are permitted for those entities and provides semantics

# Type Categories

C++ organizes the language types in two main categories:

- **Fundamental types** (often called *primitive types*): Types provided by the language itself and don't require additional headers
  - *Arithmetic types*: integer and floating point
  - `void`
  - `nullptr` C++11
- **Compound types**: Composition or references to other types
  - Pointers
  - References
  - Enumerators
  - Arrays
  - `struct`, `class`, `union`
  - Functions

C++ types can be also classified based on their properties:

- **Objects:**

- *size*: `sizeof` is defined
- *alignment requirement*: `alignof` is defined
- *storage duration*: describe when an object is allocated and deallocated
- *lifetime*, bounded by storage duration or temporary
- *value*, potentially indeterminate
- optionally, a *name*.

Types: Arithmetic, Pointers and `nullptr`, Enumerators, Arrays, `struct`,  
`class`, `union`

- **Scalar:**

- *Hold a single value* and is not composed of other objects
- *Trivially Copyable*: can be copied bit for bit
- *Standard Layout*: compatible with C functions and structs
- *Implicit Lifetime*: no user-provided constructor or destructor

Types: Arithmetic, Pointers and `nullptr` , Enumerators

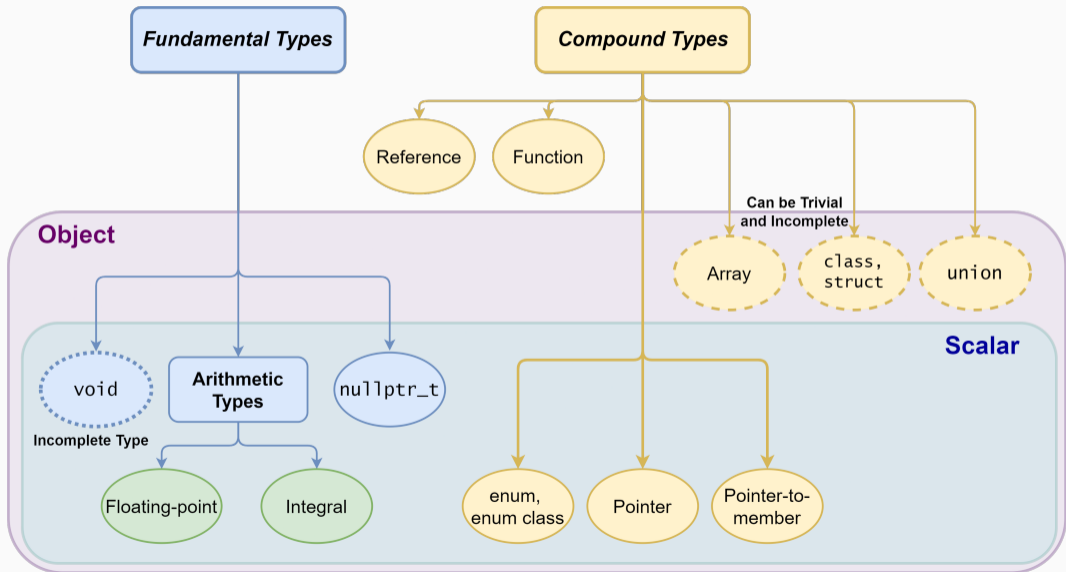
- **Trivial types**: Trivial default/copy constructor, copy assignment operator, and destructor → *Trivially Copyable*

Types: Scalar, trivial class types, arrays of such types

- **Incomplete types**: A type that has been declared but not yet defined

Types: `void` , incompletely-defined object types, e.g. `struct A;` , array of elements of incomplete type

# C++ Types Summary



# Fundamental Types

## Overview

---

## Arithmetic Types - Integral

Native Type	Bytes	Range	Fixed width types
			<stdint>
bool	1	true, false	
char †	1	implementation defined	
signed char	1	-128 to 127	int8_t
unsigned char	1	0 to 255	uint8_t
short	2	$-2^{15}$ to $2^{15}-1$	int16_t
unsigned short	2	0 to $2^{16}-1$	uint16_t
int	4	$-2^{31}$ to $2^{31}-1$	int32_t
unsigned int	4	0 to $2^{32}-1$	uint32_t
long int	4/8		int32_t/int64_t
long unsigned int	4/8*		uint32_t/uint64_t
long long int	8	$-2^{63}$ to $2^{63}-1$	int64_t
long long unsigned int	8	0 to $2^{64}-1$	uint64_t

\* 4 bytes on Windows64 systems, † signed/unsigned, two-complement from C++11

# Arithmetic Types - Floating-Point

Native Type	IEEE	Bytes	Range	Fixed width types C++23 <code>&lt;stdfloat&gt;</code>
(bfloat16)	N	2	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{+38}$	<code>std::bfloat16_t</code>
(float16)	Y	2	0.00006 to 65,536	<code>std::float16_t</code>
float	Y	4	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{+38}$	<code>std::float32_t</code>
double	Y	8	$\pm 2.23 \times 10^{-308}$ to $\pm 1.8 \times 10^{+308}$	<code>std::float64_t</code>

## Arithmetic Types - Short Name

Signed Type	short name
signed char	/
signed short int	short
signed int	int
signed long int	long
signed long long int	long long

Unsigned Type	short name
unsigned char	/
unsigned short int	unsigned short
unsigned int	unsigned
unsigned long int	unsigned long
unsigned long long int	unsigned long long

## Arithmetic Types - Suffix (Literals)

Type	SUFFIX	Example	Notes
<code>int</code>	<code>/</code>	<code>2</code>	
<code>unsigned int</code>	<code>u, U</code>	<code>3u</code>	
<code>long int</code>	<code>l, L</code>	<code>8L</code>	
<code>long unsigned</code>	<code>ul, UL</code>	<code>2ul</code>	
<code>long long int</code>	<code>ll, LL</code>	<code>4ll</code>	
<code>long long unsigned int</code>	<code>ull, ULL</code>	<code>7ULL</code>	
<code>float</code>	<code>f, F</code>	<code>3.0f</code>	only decimal numbers
<code>double</code>		<code>3.0</code>	only decimal numbers

C++23 Type	SUFFIX	Example	Notes
<code>std::bfloat16_t</code>	<code>bf16, BF16</code>	<code>3.0bf16</code>	only decimal numbers
<code>std::float16_t</code>	<code>f16, F16</code>	<code>3.0f16</code>	only decimal numbers
<code>std::float32_t</code>	<code>f32, F32</code>	<code>3.0f32</code>	only decimal numbers
<code>std::float64_t</code>	<code>f64, F64</code>	<code>3.0f64</code>	only decimal numbers
<code>std::float128_t</code>	<code>f128, F128</code>	<code>3.0f128</code>	only decimal numbers

## Arithmetic Types - Prefix (Literals)

Representation	PREFIX	Example
Binary C++14	0b	0b010101
Octal	0	0307
Hexadecimal	0x or 0X	0xFFA010

C++14 also allows *digit separators* for improving the readability `1'000'000`

## Other Arithmetic Types

- C++ also provides `long double` (no IEEE-754) of size 8/12/16 bytes depending on the implementation
- Reduced precision floating-point supports before C++23:
  - Some compilers provide support for *half* (16-bit floating-point) (GCC for ARM: `__fp16`, LLVM compiler: `half`)
  - Some modern CPUs and GPUs provide *half* instructions
  - Software support: OpenGL, Photoshop, Lightroom, `half.sourceforge.net`
- C++ does not provide **128-bit integers** even if some architectures support it. `clang` and `gcc` allow 128-bit integers as compiler extension (`__int128`)

# void Type

`void` is an incomplete type (not defined) without a value

- `void` indicates also a function with no return type or no parameters  
e.g. `void f()`, `f(void)`
- In C `sizeof(void) == 1` (GCC), while in C++ `sizeof(void)` does not compile!!

```
int main() {  
    // sizeof(void); // compile error  
}
```

## nullptr Keyword

C++11 introduces the keyword `nullptr` to represent a null pointer ( `0x0` ) and replacing the `NULL` macro

`nullptr` is an object of type `nullptr_t` → safer

```
int* p1 = NULL;      // ok, equal to int* p1 = 0l
int* p2 = nullptr;  // ok, nullptr is convertible to a pointer

int  n1 = NULL;     // ok, we are assigning 0 to n1
//int n2 = nullptr; // compile error nullptr is not convertible to an integer

//int* p2 = true ? 0 : nullptr; // compile error incompatible types
```

# Conversion Rules

---

## Conversion Rules

**Implicit type conversion rules**, applied in order, before any operation:

⊗: any operation (\*, +, /, -, %, etc.)

**(A) Floating point promotion**

`floating_type ⊗ integer_type → floating_type`

**(B) Implicit integer promotion**

`small_integral_type := any signed/unsigned integral type smaller than int`

`small_integral_type ⊗ small_integral_type → int`

**(C) Size promotion**

`small_type ⊗ large_type → large_type`

**(D) Sign promotion**

`signed_type ⊗ unsigned_type → unsigned_type`

## Examples and Common Errors

```
float    f = 1.0f;
unsigned u = 2;
int      i = 3;
short    s = 4;
uint8_t  c = 5; // unsigned char

f * u; // float × unsigned → float: 2.0f
s * c; // short × unsigned char → int: 20
u * i; // unsigned × int → unsigned: 6u
+c;    // unsigned char → int: 5
```

Integers are not floating points!

```
int  b = 7;
float a = b / 2; // a = 3 not 3.5!!
int  c = b / 2.0; // again c = 3 not 3.5!!
```

# Implicit Promotion

Integral data types smaller than 32-bit are *implicitly* promoted to `int`, independently if they are *signed* or *unsigned*

- Unary `+`, `-`, `~` and Binary `+`, `-`, `&`, `etc.` promotion:

```
char a = 48;      // '0'
cout << a;        // print '0'
cout << +a;       // print '48'
cout << (a + 0);  // print '48'

uint8_t a1 = 255;
uint8_t b1 = 255;
cout << (a1 + b1); // print '510' (no overflow)
```

auto **Keyword**

---

**C++11** The `auto` keyword specifies that the type of the variable will be automatically deduced by the compiler (from its initializer)

```
auto a = 1 + 2;    // 1 is int, 2 is int, 1 + 2 is int!  
//    -> 'a' is "int"  
auto b = 1 + 2.0; // 1 is int, 2.0 is double. 1 + 2.0 is double  
//    -> 'b' is "double"
```

`auto` can be very useful for maintainability and for hiding complex type definitions

```
for (auto i = k; i < size; i++)  
    ...
```

On the other hand, it may make the code less readable if excessively used because of type hiding

Example: `auto x = 0;` in general makes no sense (`x` is `int`)

In C++11/C++14, `auto` (as well as `decltype`) can be used to define function output types

```
auto g(int x) -> int { return x * 2; } // C++11
// "-> int" is the deduction type
// a better way to express it is:

auto g2(int x) -> decltype(x * 2) { return x * 2; } // C++11

auto h(int x) { return x * 2; } // C++14

//-----

int x = g(3); // C++11
```

In C++20, `auto` can be also used to define function input

```
void f(auto x) {}  
// equivalent to templates  
  
//-----  
  
f(3);    // 'x' is int  
f(3.0); // 'x' is double
```

# C++ Operators

---

# Operators Overview

Precedence	Operator	Description	Associativity
1	a++ a--	Suffix/postfix increment and decrement	Left-to-right
2	+a -a ++a --a ! not ~	Plus/minus, Prefix increment/decrement, Logical/Bitwise Not	Right-to-left
3	a*b a/b a%b	Multiplication, division, and remainder	Left-to-right
4	a+b a-b	Addition and subtraction	Left-to-right
5	<< >>	Bitwise left shift and right shift	Left-to-right
6	< <= > >=	Relational operators	Left-to-right
7	== !=	Equality operators	Left-to-right
8	&	Bitwise AND	Left-to-right
9	^	Bitwise XOR	Left-to-right
10		Bitwise OR	Left-to-right
11	&& and	Logical AND	Left-to-right
12	or	Logical OR	Left-to-right
13	= += -= *= /= %= <<= >>= &= ^=  =	Assignment and Compound operators	Right-to-left

Operators precedence ↗:

- **Unary** operators have higher precedence than **binary operators**
- **Standard math operators** (+, \*, etc.) have higher precedence than **comparison, bitwise, and logic** operators
- **Bitwise** and **logic** operators have higher precedence than **comparison** operators
- **Bitwise** operators have higher precedence than **logic** operators
- **Compound assignment** operators +=, -=, \*=, /=, %=, ^=, !=, &=, >>=, <<= have lower priority
- The **comma** operator has the lowest precedence (see next slides)

Examples:

```
a + b * 4;           // a + (b * 4)
a * b / c % d;      // ((a * b) / c) % d
a + b < 3 >> 4;      // (a + b) < (3 >> 4)
a && b && c || d;      // (a && b && c) || d
a and b and c or d; // (a && b && c) || d
a | b & c || e && d;  // ((a | (b & c)) || (e && d))
```

**Important:** sometimes parenthesis can make an expression verbose... but they can help!

# Prefix/Postfix Increment Semantic

**Prefix Increment/Decrement** `++i` , `-i`

- (1) Update the value
- (2) Return the new (updated) value

**Postfix Increment/Decrement** `i++` , `i--`

- (1) Save the old value (temporary)
- (2) Update the value
- (3) Return the old (original) value

Prefix/Postfix increment/decrement semantic applies not only to built-in types but also to objects

# Operation Ordering Undefined Behavior ★

Expressions with undefined (implementation-defined) behavior:

```
int i = 0;
i = ++i + 2;      // until C++11: undefined behavior
                  // since C++11: i = 3

i = 0;
i = i++ + 2;     // until C++17: undefined behavior
                  // since C++17: i = 3

f(i = 2, i = 1); // until C++17: undefined behavior
                  // since C++17: i = 2

i = 0;
a[i] = ++i;     // until C++17: undefined behavior
                  // since C++17: a[1] = 1

f(++i, ++i);    // undefined behavior
i = ++i + i++;  // undefined behavior
```

## Assignment, Compound, and Comma Operators

**Assignment** and **compound assignment** operators have *right-to-left associativity* and their expressions return the assigned value

```
int y = 2;
int x = y = 3; // y=3, then x=3
               // the same of x = (y = 3)
if (x = 4)    // assign x=4 and evaluate to true
```

The **comma operator**★ has *left-to-right associativity*. It evaluates the left expression, discards its result, and returns the right expression

```
int a = 5, b = 7;
int x = (3, 4); // discards 3, then x=4
int y = 0;
int z;
z = y, x;      // z=y (0), then returns x (4)
```

## Spaceship Operator `<=>` ★

C++20 provides the **three-way comparison operator** `<=>`, also called *spaceship operator*, which allows comparing two objects similarly of `strcmp`. The operator returns an object that can be directly compared with a positive, 0, or negative integer value

```
(3 <=> 5)      == 0; // false
('a' <=> 'a') == 0; // true

(3 <=> 5)      < 0; // true
(7 <=> 5)      < 0; // false
```

The semantic of the *spaceship operator* can be extended to any object (see next lectures) and can greatly simplify the comparison operators overloading

# Safe Comparison Operators ★

C++20 introduces a set of functions `<utility>` to safely compare integers of different types (signed, unsigned)

```
bool cmp_equal(T1 a, T2 b)
bool cmp_not_equal(T1 a, T2 b)
bool cmp_less(T1 a, T2 b)
bool cmp_greater(T1 a, T2 b)
bool cmp_less_equal(T1 a, T2 b)
bool cmp_greater_equal(T1 a, T2 b)
```

example:

```
#include <utility>
unsigned a = 4;
int      b = -3;
bool     v1 = (a > b);           // false!!!, see next slides
bool     v2 = std::cmp_greater(a, b); // true
```