

Modern C++ Programming

3. BASIC CONCEPTS I

TYPE SYSTEM, FUNDAMENTAL TYPES, AND OPERATORS

Federico Busato

2026-01-06

Table of Contents

1 The C++ Type System

- Type Categories
- Type Properties ★

2 Fundamental Types Overview

- Arithmetic Types
- Non-Standard Arithmetic Types
- `void` Type
- `nullptr`

Table of Contents

3 auto Keyword

4 C++ Operators

- Operators Precedence
- Prefix/Postfix Increment/Decrement Semantic
- Assignment, Compound, and Comma Operators
- Spaceship Operator $\langle=\rangle \star$

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*, or less precisely *builtin types*.
Types provided by the language itself that don't require additional headers
 - *Arithmetic types*: integer and floating point
 - `void`
 - `nullptr_t` 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:

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

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

- **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

examples: Arithmetic, Pointers and `nullptr`, Enumerators

- **Objects:**

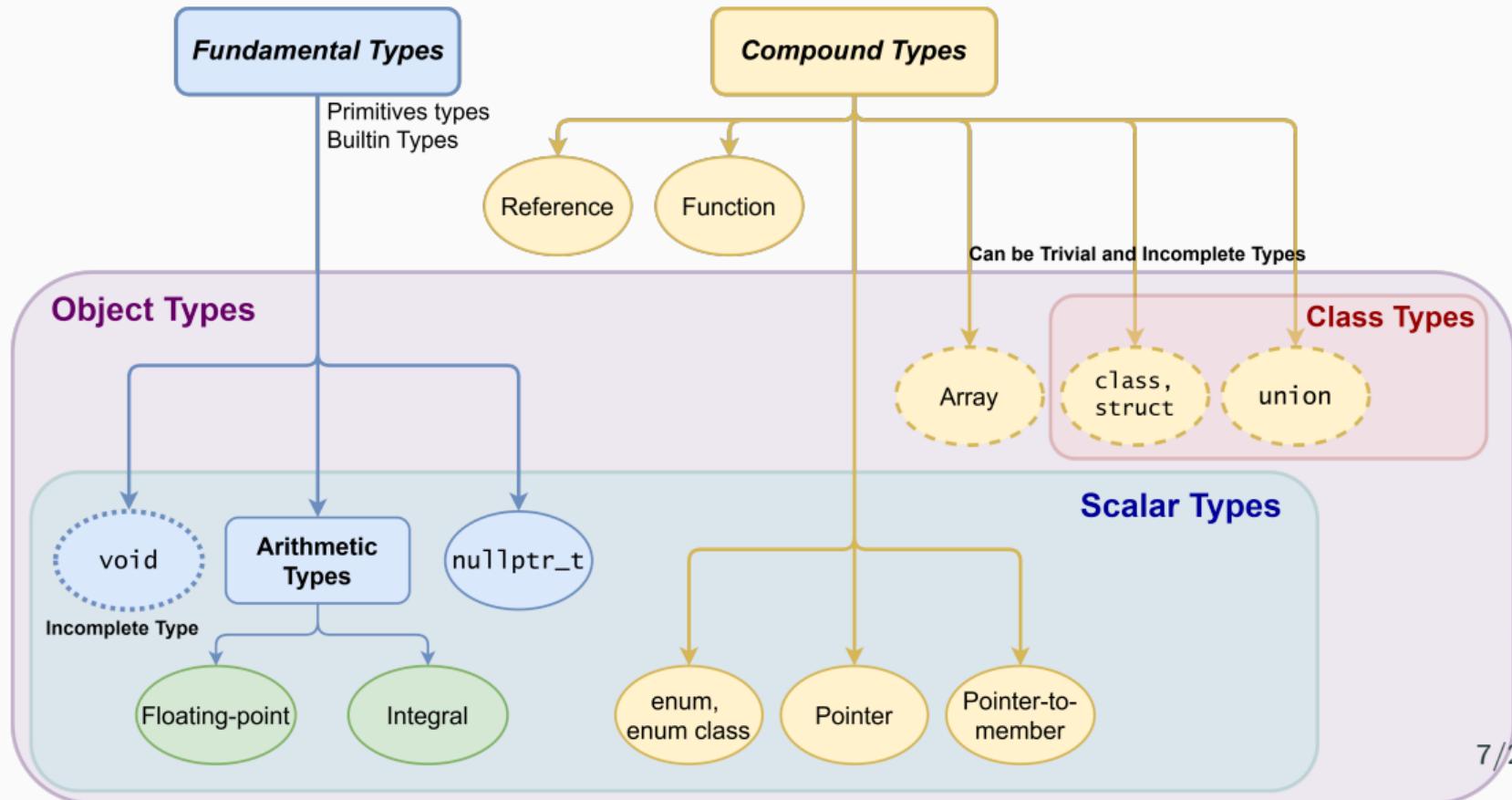
- *size*: `sizeof` is defined
- *alignment requirement*: `alignof` is defined
- *storage duration*: describe when an object is allocated and deallocated
- *lifetime*, bounded by storage duration or temporary
- *value*, potentially indeterminate
- optionally, a *name*.

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

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

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

C++ Types Summary



Fundamental Types

Overview

Arithmetic Types

Type	Bytes
bool	1
char ^{†\$}	1
unsigned char	1
short ^{\$}	2
unsigned short	2
int ^{\$}	4
unsigned	4
long ^{\$}	4*/8
long unsigned	4*/8
long long ^{\$}	8
long long unsigned	8

Standard	Type	Bytes
C++23	(bfloating16)	2
C++23	(float16)	2
	float	4
	double	8
C++23	(float128)	16

*on Windows 64-bit

Non-Standard 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
```

auto Keyword

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

```
auto a = 1 + 2; // 'a' is "int"  
auto b = 2.0;   // 'b' is double
```

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

```
// 'i' has the same type of 'k'  
for (auto i = b; i < size; i++)  
    std::vector<int> x{1, 2, 3};  
  
std::vector<int>::iterator i1 = x.begin();  
  
auto i2 = x.begin();
```

On the other hand, it may make the code less readable or even bug-prone if excessively used because of type hiding

Example: `auto x = 0;` is less readable than `int x = 0`

In C++14, `auto` (as well as `decltype`) can be used to define function output types (aka *trailing return type*)

```
auto h(int x) { return x * 2; }
```

In C++11, the return type needs to be explicitly specified:

```
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
```

In C++14, `auto` can be used to define *lambda expression* inputs

```
auto lambda = [](auto x) { return x; }
```

In C++17, `auto` is used for *structure binding*

```
int array[2] = {2, 3};  
auto [a, b] = array; // a=2, b=3
```

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

```
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 ! ~	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	&&	Logical AND	Left-to-right
12		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 *

Reading and modifying a variable within a single expression is bug prone because it can result in undefined (implementation-defined) behavior:

```
int i = 0;  
i = ++i + 2;      // since C++11: i = 3, before: undefined behavior  
  
i = 0;  
i = i++ + 2;      // since C++17: i = 3, before: undefined behavior  
  
a[i] = ++i;       // since C++17: a[1] = 1, before: undefined behavior  
  
f(i = 2, i = 1); // undefined behavior  
i = ++i + i++;   // undefined behavior
```

-Wunsequenced raises a warning when multiple *unsequenced* modifications are made on a single variable

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