Modern C++ Programming

2. Basic Concepts I

Fundamental Types and Operators

Federico Busato

2024-02-12

Table of Contents

Table of Contents

Table of Contents

Preparation

What Compiler Should I Use?

Most popular compilers:

- Microsoft Visual Code (**MSVC**) is the compiler offered by Microsoft
- The GNU Compiler Collection (**GCC**) contains the most popular C++ Linux compiler
- **Clang** is a C++ compiler based on LLVM Infrastructure available for Linux/Windows/Apple (default) platforms

<u>Suggested compiler on Linux for beginner</u>: **Clang**

- Comparable performance with GCC/MSVC and low memory usage
- Expressive diagnostics (examples and propose corrections)
- Strict C++ compliance. GCC/MSVC compatibility (inverse direction is not ensured)
- Includes very useful tools: memory sanitizer, static code analyzer, automatic formatting, linter, etc.

Install the Compiler on Linux

Install the last `gcc/g++` `(v11)` (v12 on Ubuntu 22.04)

```
$ sudo add-apt-repository ppa:ubuntu-toolchain-r/test
$ sudo apt update
$ sudo apt install gcc-12 g++-12
$ gcc-12 --version
```

Install the last `clang/clang++` `(v17)`

```
$ bash -c "$(wget -O - https://apt.llvm.org/llvm.sh)"
$ wget https://apt.llvm.org/llvm.sh
$ chmod +x llvm.sh
$ sudo ./llvm.sh 17
$ clang++ --version
```

Install the Compiler on Windows

Microsoft Visual Studio

- Direct Installer: `Visual Studio Community 2022`

Clang on Windows

Two ways:

- Windows Subsystem for Linux (WSL)
  - `Run` → `optionalfeatures`
  - Select `Windows Subsystem for Linux`, `Hyper-V`, `Virtual Machine Platform`
  - `Run` → `ms-windows-store:` → Search and install `Ubuntu 22.04 LTS`
- Clang + MSVC Build Tools
  - Download Build Tools per Visual Studio
  - Install `Desktop development with C++`

What Editor/IDE Compiler Should I Use? 1/3

Popular C++ IDE (Integrated Development Environment):

- **Microsoft Visual Studio** (MSVC) (link). Most popular IDE for Windows
- **Clion** (link). (free for student). Powerful IDE with a lot of options
- **QT-Creator** (link). Fast (written in C++), simple
- **XCode**. Default on Mac OS
- **Cevelop** (Eclipse) (link)

Standalone GUI-based coding editors:

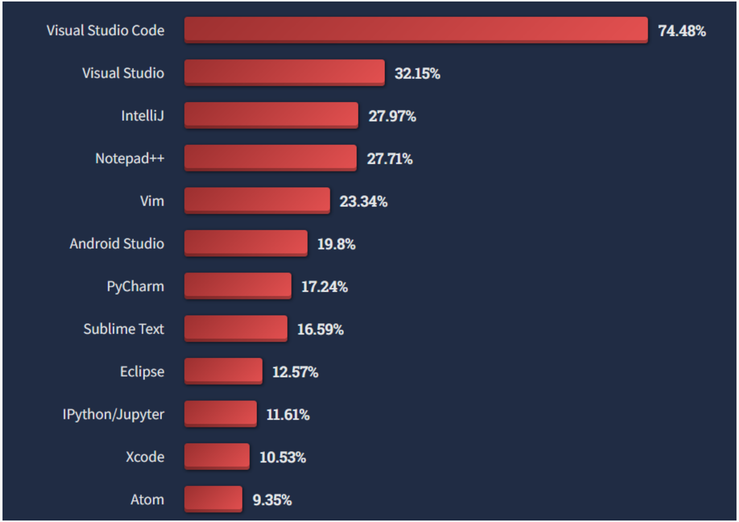
- **Microsoft Visual Studio Code** (VSCode) (link)
- **Sublime** (link)
- **Lapce** (link)

What Editor/IDE Compiler Should I Use? 2/3

Standalone text-based coding editors (powerful, but needs expertise):

- **Vim**
- **Emacs**
- **NeoVim** (link)
- **Helix** (link)

Not suggested: `Notepad, Gedit, and other similar editors` (lack of support for programming)



StackOverflow Developer Survey 2022

How to Compile?

Compile C++11, C++14, C++17, C++20 programs:

```
g++  -std=c++11 <program.cpp> -o program
g++  -std=c++14 <program.cpp> -o program
g++  -std=c++17 <program.cpp> -o program
g++  -std=c++20 <program.cpp> -o program
```

Any C++ standard is backward compatible

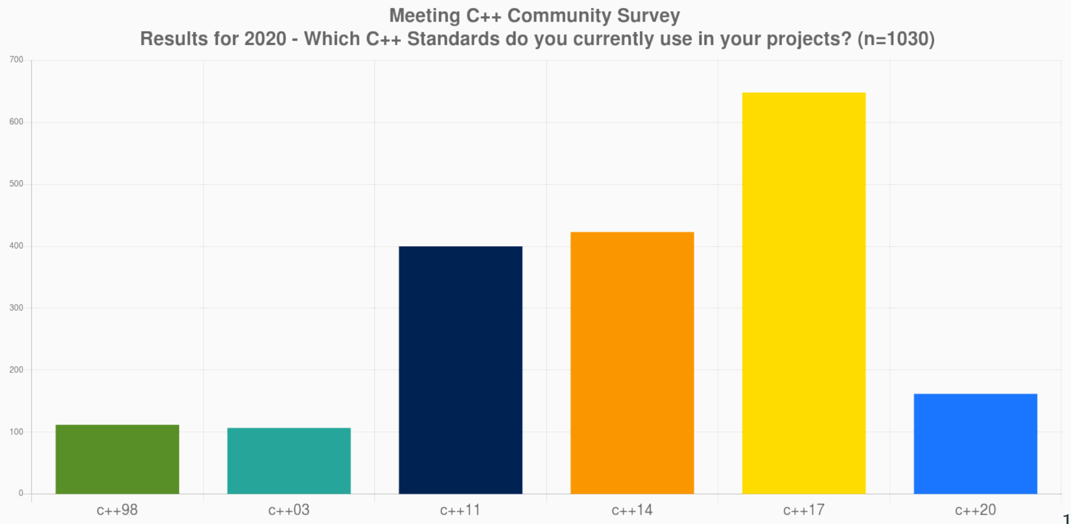
C++ is also backward compatible with C (even for very old code) except if it contains C++ keywords (`new`, `template`, `class`, `typename`, `etc.`)

We can potentially compile a pure C program in C++20

Compiler	C++11		C++14		C++17		C++20	
	Core	Library	Core	Library	Core	Library	Core	Library
`g++`	`4.8.1`	`5.1`	`5.1`	`5.1`	`7.1`	`9.0`	`11+`	`11+`
`clang++`	`3.3`	`3.3`	`3.4`	`3.5`	`5.0`	`11.0`	`16+`	`16+`
`MSVC`	`19.0`	`19.0`	`19.10`	`19.0`	`19.15`	`19.15`	`19.29+`	`19.29`

`en.cppreference.com/w/cpp/compiler_support`

C++ Standard


Meeting C++ Community Survey
Results for 2020 - Which C++ Standards do you currently use in your projects? (n=1030)
700
600
500
400
300
200
100
0
c++98
c++03
c++11
c++14
c++17
c++20

Hello World

Hello World

C code with `printf`:

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
}
```

`printf`

prints on standard output

C++ code with `streams`:

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World!\n";
}
```

`cout`

represents the standard output stream

The previous example can be written with the global `std` namespace:

```
#include <iostream>

using namespace std;
int main() {
    cout << "Hello World!\n";
}
```

Note: For sake of space and for improving the readability, we intentionally omit the `std` namespace in most slides `std::cout` is an example of *output* stream. Data is redirected to a destination, in this case the destination is the standard output

C:

```c
#include <stdio.h>
int main() {
    int    a    = 4;
    double b    = 3.0;
    char   c[] = "hello";
    printf("%d %f %s\n", a, b, c);
}
```

C++:

```cpp
#include <iostream>
int main() {
    int    a    = 4;
    double b    = 3.0;
    char   c[] = "hello";
    std::cout << a << " " << b << " " << c << "\n";
}
```

- **Type-safe**: The type of object provided to the I/O stream is known statically by the compiler. In contrast, `printf` uses `%` fields to figure out the types dynamically

- **Less error prone**: With I/O Stream, there are no redundant `%` tokens that have to be consistent with the actual objects passed to I/O stream. Removing redundancy removes a class of errors

- **Extensible**: The C++ I/O Stream mechanism allows new user-defined types to be passed to I/O stream without breaking existing code

- **Comparable performance**: If used correctly may be faster than C I/O (`printf` , `scanf` , etc.) .

I/O Stream (Common C errors) 3/3

- Forget the number of parameters:

```
printf("long phrase %d long phrase %d", 3);
```

- Use the wrong format:

```
int a = 3;
...many lines of code...
printf(" %f", a);
```

- The `%c` conversion specifier does not automatically skip any leading white space:

```
scanf("%d", &var1);
scanf(" %c", &var2);
```

std::print

C++23 introduces an improved version of `printf` function `std::print` based on *formatter strings* that provides all benefits of C++ stream and is less verbose

```
#include <print>

int main() {
    std::print("Hello World! {}, {}, {}\n", 3, 4ll, "aa");
    // print "Hello World! 3 4 aa"
}
```

This will be the default way to print when the C++23 standard is widely adopted

Fundamental Types Overview

Arithmetic Types - Integral

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

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
`int`	/	2	
`unsigned int`	u, U	3u	
`long int`	l, L	8L	
`long unsigned`	ul, UL	2ul	
`long long int`	ll, LL	4ll	
`long long unsigned int`	ull, ULL	7ULL	
`float`	f, F	`3.0f`	only decimal numbers
`double`		`3.0`	only decimal numbers

C++23 Type	SUFFIX	Example	Notes
`std::bfloat16_t`	bf16, BF16	`3.0bf16`	only decimal numbers
`std::float16_t`	f16, F16	`3.0f16`	only decimal numbers
`std::float32_t`	f32, F32	`3.0f32`	only decimal numbers
`std::float64_t`	f64, F64	`3.0f64`	only decimal numbers
`std::float128_t`	f128, F128	`3.0f128`	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 new keyword `nullptr` to represent a `null` pointer (`0x0`) and replacing the `NULL` macro

```
int* p1 = NULL;         // ok, equal to int* p1 = 0l
int* p2 = nullptr;      // ok, nullptr is a pointer not a number

int n1 = NULL;          // ok, we are assigning 0 to n1
// int n2 = nullptr; // compile error we are assigning
//                      a null pointer to an integer variable

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

Remember: `nullptr` is not a pointer, but an object of type `nullptr_t` → safer

Fundamental Types Summary

The *fundamental types*, also called *primitive* or *built-in*, are organized into three main categories:

- Integers
- Floating-points
- `void`, `nullptr`

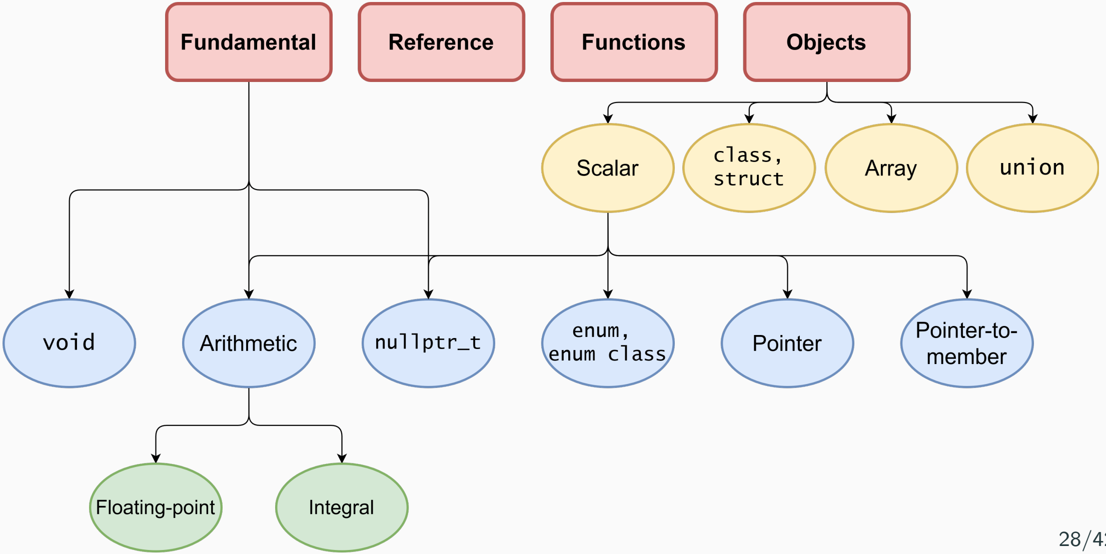
Any other entity in C++ is

- an *alias* to the correct type depending on the context and the architectures
- a *composition* of built-in types: `struct/class`, `array`, `union`

`en.cppreference.com/w/cpp/language/types`

`en.cppreference.com/w/cpp/types/integer`

C++ Types Summary



Conversion Rules

Conversion Rules

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

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

(A) Floating point promotion

`floating_type` $\otimes$ `integer_type` $\rightarrow$ `floating_type`

(B) Implicit integer promotion

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

`small_integral_type` $\otimes$ `small_integral_type` $\rightarrow$ `int`

(C) Size promotion

`small_type` $\otimes$ `large_type` $\rightarrow$ `large_type`

(D) Sign promotion

`signed_type` $\otimes$ `unsigned_type` $\rightarrow$ `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` Declaration

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`)

`auto` Keyword - Functions ★

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 but less expensive at compile-time

//--------------------------------------------------------------

f(3);    // 'x' is int
f(3.0);  // 'x' is double
```

C++ Operators

Operators Overview

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

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)

en.cppreference.com/w/cpp/language/operator_precedence

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:

```cpp
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
```

`How to compare signed and unsigned integers in C++20?`