

10.6 — Explicit type conversion (casting) and `static_cast`

👤 ALEX¹ ⌚ MARCH 4, 2025

In lesson [10.1 -- Implicit type conversion](https://www.learncpp.com/cpp-tutorial/implicit-type-conversion/) (<https://www.learncpp.com/cpp-tutorial/implicit-type-conversion/>)², we discussed that the compiler can use implicit type conversion to convert a value from one data type to another. When you want to numerically promote a value from one data type to a wider data type, using implicit type conversion is fine.

Many new C++ programmers try something like this:

```
1 | double d = 10 / 4; // does integer division, initializes d with value 2.0
```

Because `10` and `4` are both of type `int`, integer division is performed, and the expression evaluates to `int` value `2`. This value then undergoes numeric conversion to `double` value `2.0` before being used to initialize variable `d`. Most likely, this isn't what was intended.

In the case where you are using literal operands, replacing one or both of the integer literals with double literals will cause floating point division to happen instead:

```
1 | double d = 10.0 / 4.0; // does floating point division, initializes d with value 2.5
```

But what if you are using variables instead of literals? Consider this case:

```
1 | int x { 10 };  
2 | int y { 4 };  
3 | double d = x / y; // does integer division, initializes d with value 2.0
```

Because integer division is used here, variable `d` will end up with the value of `2.0`. How do we tell the compiler that we want to use floating point division instead of integer division in this case? Literal suffixes can't be used with variables. We need some way to convert one (or both) of the variable operands to a floating point type, so that floating point division will be used instead.

Fortunately, C++ comes with a number of different **type casting operators** (more commonly called **casts**) that can be used by the programmer to have the compiler perform type conversion. Because casts are explicit requests by the programmer, this form of type conversion is often called an **explicit type conversion** (as opposed to implicit type conversion, where the compiler performs a type conversion automatically).

Type casting

C++ supports 5 different types of casts: `static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`, and C-style casts. The first four are sometimes referred to as **named casts**.

For advanced readers

Cast	Description	Safe?
<code>static_cast</code>	Performs compile-time type conversions between related types.	Yes
<code>dynamic_cast</code>	Performs runtime type conversions on pointers or references in an polymorphic (inheritance) hierarchy	Yes
<code>const_cast</code>	Adds or removes <code>const</code> .	Only for adding <code>const</code>
<code>reinterpret_cast</code>	Reinterprets the bit-level representation of one type as if it were another type	No
C-style casts	Performs some combination of <code>static_cast</code> , <code>const_cast</code> , or <code>reinterpret_cast</code> .	No

Each cast works the same way. As input, the cast takes an expression (that evaluates to a value or an object), and a target type. As output, the cast returns the result of the conversion.

Because they are the most commonly used casts, we'll cover C-style casts and `static_cast` in this lesson.

Related content

We discuss `dynamic_cast` in lesson [25.10 -- Dynamic casting](https://www.learncpp.com/cpp-tutorial/dynamic-casting/) (<https://www.learncpp.com/cpp-tutorial/dynamic-casting/>)³, after we've covered other prerequisite topics.

`const_cast` and `reinterpret_cast` should generally be avoided because they are only useful in rare cases and can be harmful if used incorrectly.

Warning

Avoid `const_cast` and `reinterpret_cast` unless you have a very good reason to use them.

C-style cast

In standard C programming, casting is done via `operator()`, with the name of the type to convert to placed inside the parentheses, and the value to convert to placed immediately to the right of the closing parenthesis. In C++, this type of cast is called a **C-style cast**. You may still see these used in code that has been converted from C.

For example:

```

1  #include <iostream>
2
3  int main()
4  {
5      int x { 10 };
6      int y { 4 };
7
8      std::cout << (double)x / y << '\n'; // C-style cast of x to double
9
10     return 0;
11 }

```

In the above program, we use a C-style cast to tell the compiler to convert `x` to a `double`. Because the left operand of `operator/` now evaluates to a floating point value, the right operand will be converted to a floating point value as well, and the division will be done using floating point division instead of integer division.

C++ also provides an alternative form of C-style cast known as a **function-style cast**, which resembles a function call:

```

1  std::cout << double(x) / y << '\n'; // // function-style cast of x to double

```

The function-style cast makes it a bit easier to tell what is being converted (as it looks like a standard function argument).

There are a couple of significant reasons that C-style casts are generally avoided in modern C++.

First, although a C-style cast appears to be a single cast, it can actually perform a variety of different conversions depending on how it is used. This can include a static cast, a const cast, or a reinterpret cast (the latter two of which we mentioned above you should avoid). A C-style cast does not make it clear which cast(s) will actual be performed, which not only makes your code that much harder to understand, but also opens the door for inadvertent misuse (where you think you're implementing a simple cast and you end up doing something dangerous instead). Often this will end up producing an error that isn't discovered until runtime.

Also, because C-style casts are just a type name, parenthesis, and variable or value, they are both difficult to identify (making your code harder to read) and even more difficult to search for.

In contrast, the named casts are easy to spot and search for, make it clear what they are doing, are limited in their abilities, and will produce a compilation error if you try to misuse them.

Best practice

Avoid using C-style casts.

For advanced readers

A C-style cast tries to perform the following C++ casts, in order:

- `const_cast`
- `static_cast`
- `static_cast`, followed by `const_cast`
- `reinterpret_cast`

- `reinterpret_cast`, followed by `const_cast`

There is one thing you can do with a C-style cast that you can't do with C++ casts: C-style casts can convert a derived object to a base class that is inaccessible (e.g. because it was privately inherited).

`static_cast` should be used to cast most values

By far the most used cast in C++ is the **static cast** operator, which is accessed via the `static_cast` keyword. `static_cast` is used when we want to explicitly convert a value of one type into a value of another type.

You've previously seen `static_cast` used to convert a `char` into an `int` so that `std::cout` prints it as an integer instead of a character:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     char c { 'a' };
6 |     std::cout << static_cast<int>(c) << '\n'; // prints 97 rather than a
7 |
8 |     return 0;
9 | }
```

To perform a static cast, we start with the `static_cast` keyword, and then place the type to convert to inside angled brackets. Then inside parenthesis, we place the expression whose value will be converted. Note how much the syntax looks like a function call to a function named `static_cast<type>()` with the expression whose value will be converted provided as an argument! Static casting a value to another type of value returns a temporary object that has been direct-initialized with the converted value.

Here's how we'd use `static_cast` to solve the problem we introduced in the introduction of this lesson:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     int x { 10 };
6 |     int y { 4 };
7 |
8 |     // static cast x to a double so we get floating point division
9 |     std::cout << static_cast<double>(x) / y << '\n'; // prints 2.5
10 |
11 |     return 0;
12 | }
```

`static_cast<double>(x)` returns a temporary `double` object containing the converted value `10.0`. This temporary is then used as the left-operand of the floating point division.

There are two important properties of `static_cast`.

First, `static_cast` provides compile-time type checking. If we try to convert a value to a type and the compiler doesn't know how to perform that conversion, we will get a compilation error.

```
1 // a C-style string literal can't be converted to an int, so the following is an
2 invalid conversion
   int x { static_cast<int>("Hello") }; // invalid: will produce compilation error
```

Second, `static_cast` is (intentionally) less powerful than a C-style cast, as it will prevent certain kinds of dangerous conversions (such as those that require reinterpretation or discarding `const`).

Best practice

Favor `static_cast` when you need to convert a value from one type to another type.

For advanced readers

Since `static_cast` uses direct initialization, any explicit constructors of the target class type will be considered when initializing the temporary object to be returned. We discuss explicit constructors in lesson [14.16 -- Converting constructors and the `explicit` keyword](https://www.learncpp.com/cpp-tutorial/14.16--Converting-constructors-and-the-explicit-keyword/) (<https://www.learncpp.com/cpp-tutorial/14.16--Converting-constructors-and-the-explicit-keyword/>)⁴.

Using `static_cast` to make narrowing conversions explicit

Compilers will often issue warnings when a potentially unsafe (narrowing) implicit type conversion is performed. For example, consider the following snippet:

```
1 int i { 48 };
2 char ch = i; // implicit narrowing conversion
```

Casting an `int` (2 or 4 bytes) to a `char` (1 byte) is potentially unsafe (as the compiler can't tell whether the integer value will overflow the range of the `char` or not), and so the compiler will typically print a warning. If we used list initialization, the compiler would yield an error.

To get around this, we can use a static cast to explicitly convert our integer to a `char`:

```
1 int i { 48 };
2
3 // explicit conversion from int to char, so that a char is assigned to variable ch
4 char ch { static_cast<char>(i) };
```

When we do this, we're explicitly telling the compiler that this conversion is intended, and we accept responsibility for the consequences (e.g. overflowing the range of a `char` if that happens). Since the output of this static cast is of type `char`, the initialization of variable `ch` doesn't generate any type mismatches, and hence no warnings or errors.

Here's another example where the compiler will typically complain that converting a `double` to an `int` may result in loss of data:

```
1 int i { 100 };
2 i = i / 2.5;
```

To tell the compiler that we explicitly mean to do this:

```
1 | int i { 100 };
2 | i = static_cast<int>(i / 2.5);
```

Related content

We discuss more uses of `static_cast` in relation to class types in lesson [14.13 -- Temporary class objects](https://www.learncpp.com/cpp-tutorial/temporary-class-objects/) (<https://www.learncpp.com/cpp-tutorial/temporary-class-objects/>)⁵.

Casting vs initializing a temporary object

Let's say we have some variable `x` that we need to convert to an `int`. There are two conventional ways we can do this:

1. `static_cast<int>(x)`, which returns a temporary `int` object *direct-initialized* with `x`.
2. `int { x }`, which creates a temporary `int` object *direct-list-initialized* with `x`.

We should avoid `int (x)`, which is a C-style cast. This will return a temporary `int` direct-initialized with the value of `x` (like we'd expect from the syntax), but it also has the other downsides mentioned in the C-style cast section (like allowing the possibility of performing a dangerous conversion).

There are (at least) three notable differences between the `static_cast` and the direct-list-initialized temporary:

1. `int { x }` uses list initialization, which disallows narrowing conversions. This is great when initializing a variable, because we rarely intend to lose data in such cases. But when using a cast, it is presumed we know what we're doing, and if we want to do a cast that might lose some data, we should be able to do that. The narrowing conversion restriction can be an impediment in this case.

Let's show an example of this, including how it can lead to platform-specific issues:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     int x { 10 };
6 |     int y { 4 };
7 |
8 |     // We want to do floating point division, so one of the operands needs to be a
9 |     floating point type
10 |    std::cout << double{x} / y << '\n'; // okay if int is 32-bit, narrowing if x is
    64-bit
    }
```

In this example, we have decided to convert `x` to a `double` so we can do floating-point division rather than integer division. On a 32-bit architecture, this will work fine (because a `double` can represent all the values that can be stored in a 32-bit `int`, so it isn't a narrowing conversion). But on a 64-bit architecture, this is not the case, so converting a 64-bit `int` to a `double` is a narrowing conversion. And since list initialization disallows narrowing conversions, this won't compile on architectures where `int` is 64-bits.

2. `static_cast` makes it clearer that we are intending to perform a conversion. Although the `static_cast` is more verbose than the direct-list-initialized alternative, in this case, that's a good thing,

as it makes the conversion easier to spot and search for. That ultimately makes your code safer and easier to understand.

3. Direct-list-initialization of a temporary only allows single-word type names. Due to a weird syntax quirk, there are several places within C++ where only single-word type names are allowed (the C++ standard calls these names “simple type specifiers”). So while `int { x }` is a valid conversion syntax, `unsigned int { x }` is not.

You can see this for yourself in the following example, which produces a compile error:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     unsigned char c { 'a' };
6 |     std::cout << unsigned int { c } << '\n';
7 |
8 |     return 0;
9 | }
```

There are simple ways to work around this, the easiest of which is to use a single-word type alias:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     unsigned char c { 'a' };
6 |     using uint = unsigned int;
7 |     std::cout << uint { c } << '\n';
8 |
9 |     return 0;
10 | }
```

But why go to the trouble when you can just `static_cast`?

For all these reasons, we generally prefer `static_cast` over direct-list-initialization of a temporary.

Best practice

Prefer `static_cast` over initializing a temporary object when a conversion is desired.

Quiz time

Question #1

What's the difference between implicit and explicit type conversion?

[Show Solution](#)(javascript:void(0))⁶



[Next lesson](#)

10.7 [Typedefs and type aliases](#)

7



[Back to table of contents](#)

8



[Previous lesson](#)

10.5 [Arithmetic conversions](#)

9

10



B

U

URL

INLINE CODE

C++ CODE BLOCK

HELP!

Leave a comment...



Name*



Email*



Notify me about replies:



POST COMMENT

🔍 Find a mistake? Leave a comment above!?

👤 Avatars from <https://gravatar.com/>¹³ are connected to your provided email address.

240 COMMENTS

Newest ▼



CMaster

🕒 July 3, 2025 1:36 pm PDT

Hi,

I am reading through this tutorial to refresh my knowledge of C++ and still I am learning out of it (congrats to the authors! :))

One thing that hit my eyes in this site:

> "There is one thing you can do with a C-style cast that you can't do with C++ casts: C-style casts can convert a derived object to a base class that is inaccessible (e.g. because it was privately inherited)."

Can someone elaborate on this? The following code, indeed, does not compile with `X_ACCESSIBILITY` `private` (while it does with `X_ACCESSIBILITY public`).

[gcc version 14.2.0 (MinGW-W64 x86_64-ucrt-posix-seh, built by Brecht Sanders, r3); -std=c++23)

```
1  #include <iostream>
2
3  #define X_ACCESSIBILITY private
4
5  class X
6  {
7  public:
8      void f(int i){ std::cout << "x" << i << std::endl; }
9  };
10
11 class Y : X_ACCESSIBILITY X
12 {
13 public:
14     void f(int i){ std::cout << "y" << i << std::endl; }
15 };
16
17
18 int main()
19 {
20
21     X x;
22     x.f(0);
23
24     Y y;
25     y.f(1);
26     ((X)y).f(2);
27
28     return 0;
29 }
```

Thanks ;)

👍 0 ➡ Reply



YFN

🕒 May 29, 2025 6:20 pm PDT

TLDR for the section: just keep using `static_cast`, which was the only type of explicit casting we had learned about so far anyway.

I'm unclear on what additional value this section added that we didn't already cover in section 4.12. I suppose we learned that other types of casting exist, but are dangerous if you don't know what you're doing. I don't see why that couldn't be covered by adding one or two paragraphs to 4.12.

👍 1 ➡ Reply



Charles Kelly

🕒 May 14, 2025 6:24 am PDT

So a function-style cast, `double(x)`, is considered a C-style cast as well and should be avoided along with normal C-style casts, `(double)x`?

👍 0 ➡ Reply



Gabe

🗨 Reply to [Charles Kelly](#)¹⁴ ⌚ May 14, 2025 8:12 pm PDT

Yes, it doesn't have the same safety features as `static_cast` and can't be found through a simple Ctrl + F search.

👍 1 ➡ Reply



Charles Kelly

🗨 Reply to [Gabe](#)¹⁵ ⌚ May 16, 2025 11:50 am PDT

Thanks!

👍 0 ➡ Reply



howard roark

⌚ March 18, 2025 6:31 am PDT

I don't get any issue (using vsc)

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    unsigned char c { 'a' };
```

```
    std::cout << unsigned int { c } << '\n';
```

```
    return 0;
```

```
}
```

👍 5 ➡ Reply



Nimantha Cooray

⌚ March 11, 2025 8:06 pm PDT

Hi,

I am trying to understand the following statement.

"On a 32-bit architecture, this will work fine (because a double can represent all the values that can be stored in a 32-bit int, so it isn't a narrowing conversion). But on a 64-bit architecture, this is not the case, so converting a 64-bit int to a double is a narrowing conversion."

So as I understand, on a 32-bit architecture, 32-bit int has 31-bits to represent the max value (ignoring the sign bit).

The double (64-bits) has 52-bits (fraction part in IEEE-754) to represent the max value of the int. So it is possible (because $32 < 52$).

But on a 64-bit architecture, 64-bit int has 63-bits to represent the max value (ignoring the sign bit). The double (64-bits) has only 52-bits (fraction part in IEEE-754) to represent the max value of the int. So it is

NOT possible (because $63 > 52$).

Is this correct?

 0  Reply



Felipe

 Reply to [Nimantha Cooray](#)¹⁶  March 31, 2025 6:38 am PDT

What you said is right. Also, look up "two's complement": C++ ints don't use sign bits.

 1  Reply



Tobito

 February 5, 2025 10:49 am PST

```
data.read(reinterpret_cast<char*>(&readStudent), sizeof(student));
```

I saw alot of code which using reinterpret cast, especially on this one. I shall need to change it to static_cast<char*> for the above code?

 0  Reply



Alex

Author

 Reply to [Tobito](#)¹⁷  February 8, 2025 8:43 pm PST

I don't think a static_cast would work at all here, as the compiler will complain that it can't find a conversion from the type of readStudent to char*.

That said, this code seems non-portable, as it's reading char data into whatever readStudent is. I would expect byte order and padding issues to occur across different platforms.

 0  Reply

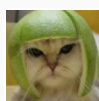


Tobito

 Reply to [Alex](#)¹⁸  February 12, 2025 10:05 am PST


so we must using reinterpret cast for this one (because static cast can not work)? or any other solution for this code?

 0  Reply



Alex

Author

 Reply to [Tobito](#)¹⁹  February 12, 2025 12:10 pm PST

Given the interface constraints, seems like you'd either have to use reinterpret_cast or a different method altogether.

 0  Reply



danumJanum

🕒 February 2, 2025 5:37 am PST

Hey man thanks for teaching C++ to people like us for 100% free. Your tutorials are helping me alot



1

➡ Reply



r.kh

🕒 November 14, 2024 2:34 pm PST

First, `static_cast` provides compile-time type checking

Is this also a property of C-style casting? If so, it is not a unique advantage of using `static_cast` over other casting methods!



0

➡ Reply



Alex

Author

🗨 Reply to [r.kh](#) ²⁰ 🕒 November 15, 2024 2:24 pm PST

Didn't say it was a unique advantage.

I note that it does compile-time type checking to differentiate it from `dynamic_cast` (covered in a future lesson), which is a runtime cast.



0

➡ Reply



rkh

🗨 Reply to [Alex](#) ²¹ 🕒 November 16, 2024 3:59 am PST

```
1 // a C-style string literal can't be converted to an int, so the
2 following is an invalid conversion
  int x { static_cast<int>("Hello") }; // invalid: will produce
    compilation error
```

another issue is that this conversion is also invalid to perform a C-style cast and if I'm not mistaken there are no casts to do directly this conversion. Maybe converting an int to a pointer of type int would be a better idea although the pointer lesson will be covered latter.

```
1 | int* p = static_cast<int*>(5)
```

📝 Last edited 7 months ago by rkh



0

➡ Reply



Alex

Author

🗨 Reply to [rkh](#) ²² 🕒 November 18, 2024 11:55 am PST

Why would you ever want to cast a pointer to a char array to an int? In C++, you can do this with a `reinterpret_cast`. But I think it will only work when `sizeof(const char*) ==`

`sizeof(int)`, which probably isn't the case on 64-bit systems.

👍 0

↩ Reply



rkh

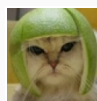
↩ Reply to [Alex](#)²³ 🕒 November 19, 2024 1:27 am PST

I would prefer not to do this conversion. My talk focuses on the differences between C-style casts and `static_cast`, which is the emphasis of this lesson. since this example is not performed for two casting cases, it does not illustrate the advantages of using `static_cast` over a C-style cast.

I apologize if this suggestion is incorrect, and I appreciate your clarification. Thank you!

👍 0

↩ Reply



Alex

Author

↩ Reply to [rkh](#)²⁴ 🕒 November 19, 2024 4:30 pm PST

The lesson tells you what kind of casts a C-style cast will try. In order: `const_cast`, `static_cast`, `static_cast` followed by `const_cast`, `reinterpret_cast`, `reinterpret_cast` followed by `const_cast`.

Therefore, the difference between a C-style cast and a `static_cast` is that a C-style cast will do a `const_cast` or a `reinterpret_cast`, whereas a `static_cast` will not.

The advantage of `static_cast` is that its safer, as we normally don't want to discard `const` or `reinterpret` types.

👍 2

↩ Reply



rkh

↩ Reply to [Alex](#)²¹ 🕒 November 16, 2024 3:23 am PST

thanks for clarification

I think that sentence seems a bit general; adding your point would enhance clarity and comprehension.

👍 0

↩ Reply



User0

🕒 October 3, 2024 4:09 pm PDT

on

```
1 | int& ref{ static_cast<int&>(x) }; // invalid: will produce compilation error
```

till this point (lesson 10.6) you have not introduced references yet, so I had find it confusing when I saw it.

📝 Last edited 9 months ago by User0

👍 0 ➡ Reply



Alex Author

🗨 Reply to [User0](#)²⁵ ⌚ October 4, 2024 5:29 pm PDT

Rewrote some of the text in the lesson and removed the example. Thanks for pointing this out.

👍 0

➡ Reply



qwerty

⌚ September 17, 2024 4:23 am PDT

isn't this a narrowing conversion ?

```
int main()
```

```
{
```

```
char c { 'a' };
```

```
std::cout << c << ' ' << static_cast<int>(c) << '\n'; // prints a 97
```

```
return 0;
```

```
}
```

my compiler treats it as an error

👍 0

➡ Reply



Alex Author

🗨 Reply to [qwerty](#)²⁶ ⌚ September 20, 2024 1:43 pm PDT

There are no narrowing conversions in this code.

`char c { 'a' }` doesn't require a conversion (Per

https://en.cppreference.com/w/cpp/language/character_literal, a character literal has type `char`)

`static_cast<int>(c)` is an widening conversion from smaller integral type `char` to a larger integral type `int`.

If you are getting an error, it may be because you forgot to `#include <iostream>`.

👍 0

➡ Reply

Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/implicit-type-conversion/>
3. <https://www.learncpp.com/cpp-tutorial/dynamic-casting/>
4. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/>

5. <https://www.learncpp.com/cpp-tutorial/temporary-class-objects/>
6. `javascript:void(0)`
7. <https://www.learncpp.com/cpp-tutorial/typedefs-and-type-aliases/>
8. <https://www.learncpp.com/>
9. <https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/>
10. <https://www.learncpp.com/explicit-type-conversion-casting-and-static-cast/>
11. <https://www.learncpp.com/cpp-tutorial/why-non-const-global-variables-are-evil/>
12. <https://www.learncpp.com/cpp-tutorial/scoped-enumerations-enum-classes/>
13. <https://gravatar.com/>
14. <https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/#comment-610048>
15. <https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/#comment-610068>
16. <https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/#comment-608471>
17. <https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/#comment-607408>
18. <https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/#comment-607525>
19. <https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/#comment-607657>
20. <https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/#comment-604172>
21. <https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/#comment-604203>
22. <https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/#comment-604212>
23. <https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/#comment-604269>
24. <https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/#comment-604307>
25. <https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/#comment-602675>
26. <https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/#comment-602058>
27. <https://g.ezoic.net/privacy/learncpp.com>