

10.1 — Implicit type conversion

👤 **ALEX**¹ ⌚ **MARCH 3, 2025**

We introduced type conversion in lesson [4.12 -- Introduction to type conversion and static_cast](https://www.learncpp.com/cpp-tutorial/introduction-to-type-conversion-and-static_cast/) (https://www.learncpp.com/cpp-tutorial/introduction-to-type-conversion-and-static_cast/)². To recap the most important points from that lesson:

- The process of converting data from one type to another type is called “type conversion”.
- Implicit type conversion is performed automatically by the compiler when one data type is required, but a different data type is supplied.
- Explicit type conversion is requested by using a cast operator, such as `static_cast`.
- Conversions do not change the data being converted. Instead, the conversion process uses that data as input, and produces the converted result.
- When converting a value to another type of value, the conversion process produces a temporary object of the target type that holds the result of the conversion.

In the first half of this chapter, we’re going to dig a bit deeper into how type conversion works. We’ll start with implicit conversions in this lesson, and explicit type conversions (casting) in upcoming lesson [10.6 -- Explicit type conversion \(casting\) and static_cast](https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/) (<https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/>)³. Since type conversion is used all over the place, having some understanding of what’s happening under the hood when a conversion is needed is important. This knowledge is also relevant when for understanding how overloaded functions (functions that can have the same name as other functions) work.

Author’s note

In this chapter, we’ll focus on the conversion of values to other types of values. We’ll cover other types of conversions once we introduce the prerequisite topics (such as pointers, references, inheritance, etc...).

Why conversions are needed

The value of an object is stored as a sequence of bits, and the data type of the object tells the compiler how to interpret those bits into meaningful values. Different data types may represent the “same” value differently. For example, the integer value `3` might be stored as binary `0000 0000 0000 0000 0000 0000 0000 0011`, whereas floating point value `3.0` might be stored as binary `0100 0000 0100 0000 0000 0000 0000 0000`.

So what happens when we do something like this?

```
1 | float f{ 3 }; // initialize floating point variable with int 3
```

In such a case, the compiler can't just copy the bits used to represent `int` value `3` into the memory allocated for `float` variable `f`. If it were to do so, then when `f` (which has type `float`) was evaluated, those bits would be interpreted as a `float` rather than an `int`, and who knows what `float` value we'd end up with!

As an aside...

The following program actually prints `int` value `3` as if it were a `float`:

```
1 | #include <iostream>
2 | #include <cstring>
3 |
4 | int main()
5 | {
6 |     int n { 3 };           // here's int value 3
7 |     float f {};           // here's our float variable
8 |     std::memcpy(&f, &n, sizeof(float)); // copy the bits from n into f
9 |     std::cout << f << '\n'; // print f (containing the bits from n)
10 |
11 |     return 0;
12 | }
```

This produces the following result:

```
4.2039e-45
```

Instead, the integer value `3` needs to be converted into the equivalent floating point value `3.0`, which can then be stored in the memory allocated for `f` (using the bit representation for `float` value `3.0`).

When implicit type conversion happens

Implicit type conversion (also called **automatic type conversion** or **coercion**) is performed automatically by the compiler when an expression of some type is supplied in a context where some other type is expected. The vast majority of type conversions in C++ are implicit type conversions. For example, implicit type conversion happens in all of the following cases:

When initializing (or assigning a value to) a variable with a value of a different data type:

```
1 | double d{ 3 }; // int value 3 implicitly converted to type double
2 | d = 6; // int value 6 implicitly converted to type double
```

When the type of a return value is different from the function's declared return type:

```
1 | float doSomething()
2 | {
3 |     return 3.0; // double value 3.0 implicitly converted to type float
4 | }
```

When using certain binary operators with operands of different types:

```
1 | double division{ 4.0 / 3 }; // int value 3 implicitly converted to type double
```

When using a non-Boolean value in an if-statement:

```
1 | if (5) // int value 5 implicitly converted to type bool
2 | {
3 | }
```

When an argument passed to a function is a different type than the function parameter:

```
1 | void doSomething(long l)
2 | {
3 | }
4 |
5 | doSomething(3); // int value 3 implicitly converted to type long
```

So how does the compiler know how to convert a value to a different type anyway?

The standard conversions

As part of the core language, the C++ standard defines a collection of conversion rules known as the “standard conversions”. The **standard conversions** specify how various fundamental types (and certain compound types, including arrays, references, pointers, and enumerations) convert to other types within that same group.

As of C++23, there are 14 different standard conversions. These can be roughly grouped into 5 general categories:

Category	Meaning	Link
Numeric promotions	Conversions of small integral types to <code>int</code> or <code>unsigned int</code> , and of <code>float</code> to <code>double</code> .	10.2 -- Floating-point and integral promotion ⁴
Numeric conversions	Other integral and floating point conversions that aren't promotions.	10.3 -- Numeric conversions ⁵
Qualification conversions	Conversions that add or remove <code>const</code> or <code>volatile</code> .	
Value transformations	Conversions that change the value category of an expression	12.2 -- Value categories (lvalues and rvalues) ⁶
Pointer conversions	Conversions from <code>std::nullptr</code> to pointer types, or pointer types to other pointer types	

For example, converting an `int` value to a `float` value falls under the numeric conversions category, so the compiler to perform such a conversion, the compiler simply need apply the `int` to `float` numeric conversion rules.

The numeric conversions and numeric promotions are the most important of these categories, and we'll cover them in more detail in upcoming lessons.

For advanced readers

Here is the full list of standard conversions:

Category	Standard Conversion	Description	Also See
Value transformation	Lvalue-to-rvalue	Converts lvalue expression to rvalue expression	12.2 -- Value categories (lvalues and rvalues) ⁶
Value transformation	Array-to-pointer	Converts C-style array to pointer to first array element (a.k.a. array decay)	17.8 -- C-style array decay ⁷
Value transformation	Function-to-pointer	Converts function to function pointer	20.1 -- Function Pointers ⁸
Value transformation	Temporary materialization	Converts value to temporary object	
Qualification conversion	Qualification conversion	Adds or removes <code>const</code> or <code>volatile</code> from types	
Numeric promotions	Integral promotions	Converts smaller integral types to <code>int</code> or <code>unsigned int</code>	10.2 -- Floating-point and integral promotion ⁴
Numeric promotions	Floating point promotions	Converts <code>float</code> to <code>double</code>	10.2 -- Floating-point and integral promotion ⁴
Numeric conversions	Integral conversions	Integral conversions that aren't integral promotions	10.3 -- Numeric conversions ⁵
Numeric conversions	Floating point conversions	Floating point conversions that aren't floating point promotions	10.3 -- Numeric conversions ⁵
Numeric conversions	Integral-floating conversions	Converts integral and floating point types	10.3 -- Numeric conversions ⁵
Numeric conversions	Boolean conversions	Converts integral, unscoped enumeration, pointer, or pointer-to-member to <code>bool</code>	4.10 -- Introduction to if statements ⁹
Pointer conversions	Pointer conversions	Converts <code>std::nullptr</code> to pointer, or pointer to void pointer or base class	
Pointer conversions	Pointer-to-member conversions	Converts <code>std::nullptr</code> to pointer-to-member or pointer-to-member of base class to pointer-to-member of derived class	
Pointer conversions	Function pointer conversions	Converts pointer-to-noexcept-function to pointer-to-function	

Type conversion can fail

When a type conversion is invoked (whether implicitly or explicitly), the compiler will determine whether it can convert the value from the current type to the desired type. If a valid conversion can be found, then the

compiler will produce a new value of the desired type.

If the compiler can't find an acceptable conversion, then the compilation will fail with a compile error. Type conversions can fail for any number of reasons. For example, the compiler might not know how to convert a value between the original type and the desired type.

For example:

```
1 | int main()
2 | {
3 |     int x { "14" };
4 |
5 |     return 0;
6 | }
```

The because there isn't a standard conversion from the string literal "14" to `int`, the compiler will produce an error. For example, GCC produces the error: `prog.cc:3:13: error: invalid conversion from 'const char*' to 'int' [-fpermissive]`.

In other cases, specific features may disallow some categories of conversions. For example:

```
1 | int x { 3.5 }; // brace-initialization disallows conversions that result in data loss
```

Even though the compiler knows how to convert a `double` value to an `int` value, narrowing conversions are disallowed when using brace-initialization.

There are also cases where the compiler may not be able to figure out which of several possible type conversions is the best one to use. We'll see examples of this in lesson [11.3 -- Function overload resolution and ambiguous matches](https://www.learncpp.com/cpp-tutorial/function-overload-resolution-and-ambiguous-matches/) (<https://www.learncpp.com/cpp-tutorial/function-overload-resolution-and-ambiguous-matches/>)¹⁰.

The full set of rules describing how type conversions work is both lengthy and complicated, and for the most part, type conversion "just works". In the next set of lessons, we'll cover the most important things you need to know about the standard conversions. If finer detail is required for some uncommon case, the full rules are detailed in [technical reference documentation for implicit conversions](https://en.cppreference.com/w/cpp/language/implicit_conversion) (https://en.cppreference.com/w/cpp/language/implicit_conversion)¹¹.

Let's get to it!



[Next lesson](#)

10.2 [Floating-point and integral promotion](#)

4



[Back to table of contents](#)

12



[Previous lesson](#)

9.x [Chapter 9 summary and quiz](#)

13




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/>¹⁷ are connected to your provided email address.

197 COMMENTS

Newest ▼


- 



mrme

🕒 June 14, 2025 3:59 am PDT

"The because there isn't a standard conversion from the string literal "14" to int"

mistake in the sentence, just a heads up


 0



 Reply
- 

gold

🕒 May 11, 2025 7:50 am PDT

Todd Howard- "It just works."


 1



 Reply
- 

perdix

🕒 March 30, 2025 2:03 am PDT

Hi! I'm really enjoying the tutorials so far. I think I've spotted a type as well, in the Type conversion section: "The because there isn't a standard conversion..."

 2

 Reply
- 

KALEIDOSCOPE

Hi Alex! Thanks for the awesome tutorial so far! However I think I've spotted a typo which is located on the 11th row of the table for advanced readers, "pointer-to-memver". This doesnt harm that it's a great article!

👍 0 ➡ Reply



ska00

🕒 January 14, 2025 10:39 am PST

Hey Alex! Thanks for the great tutorials.

If the compiler does the implicit conversion what is doing the conversion when we pass a function an argument that does not match the data type of its parameter? Or what I'm trying to ask is how does C++ do conversions in run-time?

👍 0 ➡ Reply



Gotcha

🕒 October 29, 2024 2:22 am PDT

Ah yes! The famous upcoming lesson `%Failed lesson reference, id 1619%`. Can't wait to get to it.

👍 7 ➡ Reply



lamb

🕒 July 12, 2024 7:42 pm PDT

why "The compiler may apply zero, one, or more than one standard conversions in the conversion process"?

> https://en.cppreference.com/w/cpp/language/implicit_conversion

> **Order of the conversions**

> Implicit conversion sequence consists of the following, in this order:

>

> 1) zero or one standard conversion sequence;

> 2) zero or one user-defined conversion;

> 3) zero or one standard conversion sequence (only if a user-defined conversion is used).

>

> When considering the argument to a constructor or to a user-defined conversion function, only one standard conversion sequence is allowed (otherwise user-defined conversions could be effectively chained). When converting from one non-class type to another non-class type, only a standard conversion sequence is allowed.

In both conversion case(class type or non-class), it says that "only one standard conversion sequence is allowed".

Maybe I misunderstood the description on cppreference. Does the 1,2,3 listed above mean that the conversion is carried out in the order of these three steps, and the conversion ends after the third step,

or does it mean that the conversion returns to the first step and repeats the cycle after the second step?

If I made any mistakes, please point them out. Thanks for your time.

 Last edited 11 months ago by [lamb](#)

 0  Reply

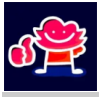


Alex Author

 Reply to [lamb](#)¹⁸  July 13, 2024 5:51 pm PDT

The 3 steps are carried out in order (with no loopback to the beginning). Step 1 can apply a standard conversion. Step 3 can also apply a standard conversion (if a user-defined conversion was applied in step 2). Therefore, we can have a standard-user-standard conversion sequence, which is two standard conversions.

 1  Reply



i forgot what name i set here

 June 5, 2024 12:44 am PDT

Will we cover dealing with inaccurate floating point rounding in the future?

 0  Reply



Alex Author

 Reply to [i forgot what name i set here](#)¹⁹  June 5, 2024 2:39 pm PDT

We already do in lesson <https://www.learncpp.com/cpp-tutorial/relational-operators-and-floating-point-comparisons/>

 2  Reply



Swaminathan R

 May 4, 2024 12:20 pm PDT

```
1 | double d{ 3 }; // int value 3 implicitly converted to type double
```

A basic(and possibly stupid) question. The symbols “3” and “3.0” only makes sense to us humans. So, how does the computer know 3 in integer corresponds to 3.0 in decimal?

 0  Reply



Alex Author

 Reply to [Swaminathan R](#)²⁰  May 8, 2024 11:44 am PDT

The compiler comes with a bunch of built-in type conversions that understand how to do things like convert integral to floating point values and vice-versa. We cover this later in this chapter.



kloderdakarma

🕒 April 13, 2024 10:03 pm PDT

Are implicit types conversion only valid for fundamental data types ?

👍 1 ➡ Reply



Alex Author

➡ Reply to [kloderdakarma](#)²¹ 🕒 April 16, 2024 4:42 pm PDT

No. Compound types and user-defined/program-defined types can be implicitly converted (if the conversion is valid).

👍 0 ➡ Reply

Links

1. <https://www.learncpp.com/author/Alex/>
2. https://www.learncpp.com/cpp-tutorial/introduction-to-type-conversion-and-static_cast/
3. <https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/>
4. <https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/>
5. <https://www.learncpp.com/cpp-tutorial/numeric-conversions/>
6. <https://www.learncpp.com/cpp-tutorial/value-categories-lvalues-and-rvalues/>
7. <https://www.learncpp.com/cpp-tutorial/c-style-array-decay/>
8. <https://www.learncpp.com/cpp-tutorial/function-pointers/>
9. <https://www.learncpp.com/cpp-tutorial/introduction-to-if-statements/>
10. <https://www.learncpp.com/cpp-tutorial/function-overload-resolution-and-ambiguous-matches/>
11. https://en.cppreference.com/w/cpp/language/implicit_conversion
12. <https://www.learncpp.com/>
13. <https://www.learncpp.com/cpp-tutorial/chapter-9-summary-and-quiz/>
14. <https://www.learncpp.com/implicit-type-conversion/>
15. <https://www.learncpp.com/cpp-tutorial/static-local-variables/>
16. <https://www.learncpp.com/cpp-tutorial/unscoped-enumerations/>
17. <https://gravatar.com/>
18. <https://www.learncpp.com/cpp-tutorial/implicit-type-conversion/#comment-599510>
19. <https://www.learncpp.com/cpp-tutorial/implicit-type-conversion/#comment-597966>
20. <https://www.learncpp.com/cpp-tutorial/implicit-type-conversion/#comment-596677>
21. <https://www.learncpp.com/cpp-tutorial/implicit-type-conversion/#comment-595745>
22. <https://g.ezoic.net/privacy/learncpp.com>

