

## 10.3 — Numeric conversions

👤 **ALEX<sup>1</sup>** ⌚ **AUGUST 20, 2024**

In the previous lesson ([10.2 -- Floating-point and integral promotion](https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/)), we covered numeric promotions, which are conversions of specific narrower numeric types to wider numeric types (typically `int` or `double`) that can be processed efficiently.

C++ supports another category of numeric type conversions, called **numeric conversions**. These numeric conversions cover additional type conversions between fundamental types.

### Key insight

Any type conversion covered by the numeric promotion rules ([10.2 -- Floating-point and integral promotion](https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/)) is called a numeric promotion, not a numeric conversion.

There are five basic types of numeric conversions.

1. Converting an integral type to any other integral type (excluding integral promotions):

```
1 | short s = 3; // convert int to short
2 | long l = 3; // convert int to long
3 | char ch = s; // convert short to char
4 | unsigned int u = 3; // convert int to unsigned int
```

2. Converting a floating point type to any other floating point type (excluding floating point promotions):

```
1 | float f = 3.0; // convert double to float
2 | long double ld = 3.0; // convert double to long double
```

3. Converting a floating point type to any integral type:

```
1 | int i = 3.5; // convert double to int
```

4. Converting an integral type to any floating point type:

```
1 | double d = 3; // convert int to double
```

5. Converting an integral type or a floating point type to a bool:

```
1 | bool b1 = 3; // convert int to bool
2 | bool b2 = 3.0; // convert double to bool
```

## As an aside...

Because brace initialization strictly disallows some types of numeric conversions (more on this next lesson), we use copy initialization in this lesson (which does not have any such limitations) in order to keep the examples simple.

## Safe and unsafe conversions

Unlike numeric promotions (which are always value-preserving and thus “safe”), many numeric conversions are unsafe. An **unsafe conversion** is one where at least one value of the source type cannot be converted into an equal value of the destination type.

Numeric conversions fall into three general safety categories:

1. *Value-preserving conversions* are safe numeric conversions where the destination type can exactly represent all possible values in the source type.

For example, `int` to `long` and `short` to `double` are safe conversions, as the source value can always be converted to an equal value of the destination type.

```
1 | int main()
2 | {
3 |     int n { 5 };
4 |     long l = n; // okay, produces long value 5
5 |
6 |     short s { 5 };
7 |     double d = s; // okay, produces double value 5.0
8 |
9 |     return 0;
10| }
```

Compilers will typically not issue warnings for implicit value-preserving conversions.

A value converted using a value-preserving conversion can always be converted back to the source type, resulting in a value that is equivalent to the original value:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     int n = static_cast<int>(static_cast<long>(3)); // convert int 3 to long and back
6 |     std::cout << n << '\n';                       // prints 3
7 |
8 |     char c = static_cast<char>(static_cast<double>('c')); // convert 'c' to double and
9 |     back
10|     std::cout << c << '\n';                       // prints 'c'
11|
12|     return 0;
   | }
```

2. *Reinterpretive conversions* are unsafe numeric conversions where the converted value may be different than the source value, but no data is lost. Signed/unsigned conversions fall into this category.

For example, when converting a `signed int` to an `unsigned int`:

```

1 | int main()
2 | {
3 |     int n1 { 5 };
4 |     unsigned int u1 { n1 }; // okay: will be converted to unsigned int 5 (value
5 |     preserved)
6 |
7 |     int n2 { -5 };
8 |     unsigned int u2 { n2 }; // bad: will result in large integer outside range of
9 |     signed int
10 |
    return 0;
    }

```

In the case of `u1`, the signed int value `5` is converted to unsigned int value `5`. Thus, the value is preserved in this case.

In the case of `u2`, the signed int value `-5` is converted to an unsigned int. Since an unsigned int can't represent negative numbers, the result will be modulo wrapped to a large integral value that is outside the range of a signed int. The value is not preserved in this case.

Such value changes are typically undesirable, and will often cause the program to exhibit unexpected or implementation-defined behavior.

## Related content

We discuss how out-of-range values are converted between signed and unsigned types 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/](https://www.learncpp.com/cpp-tutorial/introduction-to-type-conversion-and-static_cast/))<sup>3</sup>.

## Warning

Even though reinterpretive conversions are unsafe, most compilers leave implicit signed/unsigned conversion warnings disabled by default.

This is because in some areas of modern C++ (such as when working with standard library arrays), signed/unsigned conversions can be hard to avoid. And practically speaking, the majority of such conversions do not actually result in a value change. Therefore, enabling such warnings can lead to many spurious warnings for signed/unsigned conversions that are actually okay (drowning out legitimate warnings).

If you choose to leave such warnings disabled, be extra careful of inadvertent conversions between these types (particularly when passing an argument to a function taking a parameter of the opposite sign).

Values converted using a reinterpretive conversion can be converted back to the source type, resulting in a value equivalent to the original value (even if the initial conversion produced a value out of range of the source type). As such, reinterpretive conversions do not lose data during the conversion process.

```

1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     int u = static_cast<int>(static_cast<unsigned int>(-5)); // convert '-5' to
6 |     unsigned and back
7 |     std::cout << u << '\n'; // prints -5
8 |
9 |     return 0;
10 | }

```

## For advanced readers

Prior to C++20, converting an unsigned value that is out-of-range of the signed value is technically implementation-defined behavior (due to the allowance that signed integers could use a different binary representation than unsigned integers). In practice, this was a non-issue on modern system.

C++20 now requires that signed integers use 2s complement. As a result, the conversion rules were changed so that the above is now well-defined as a reinterpretive conversion (an out-of-range conversion will produce modulo wrapping).

Note that while such conversions are well defined, signed arithmetic overflow (which occurs when an arithmetic operation produces a value outside the range that can be stored) is still undefined behavior.

3. *Lossy conversions* are unsafe numeric conversions where data may be lost during the conversion.

For example, `double` to `int` is a conversion that may result in data loss:

```

1 | int i = 3.0; // okay: will be converted to int value 3 (value preserved)
2 | int j = 3.5; // data lost: will be converted to int value 3 (fractional value 0.5
   | lost)

```

Conversion from `double` to `float` can also result in data loss:

```

1 | float f = 1.2; // okay: will be converted to float value 1.2 (value preserved)
2 | float g = 1.23456789; // data lost: will be converted to float 1.23457 (precision
   | lost)

```

Converting a value that has lost data back to the source type will result in a value that is different than the original value:

```

1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     double d { static_cast<double>(static_cast<int>(3.5)) }; // convert double 3.5 to
6 |     int and back
7 |     std::cout << d << '\n'; // prints 3
8 |
9 |     double d2 { static_cast<double>(static_cast<float>(1.23456789)) }; // convert
10 |    double 1.23456789 to float and back
11 |    std::cout << d2 << '\n'; // prints 1.23457
12 |    return 0;
13 | }

```

For example, if `double` value `3.5` is converted to `int` value `3`, the fractional component `0.5` is lost. When `3` is converted back to a `double`, the result is `3.0`, not `3.5`.

Compilers will generally issue a warning (or in some cases, an error) when an implicit lossy conversion would be performed at runtime.

## Warning

Some conversions may fall into different categories depending on the platform.

For example, `int` to `double` is typically a safe conversion, because `int` is usually 4 bytes and `double` is usually 8 bytes, and on such systems, all possible `int` values can be represented as a `double`. However, there are architectures where both `int` and `double` are 8 bytes. On such architectures, `int` to `double` is a lossy conversion!

We can demonstrate this by casting a long long value (which must be at least 64 bits) to double and back:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     std::cout << static_cast<long long>(static_cast<double>(10000000000000001LL));
6 |
7 |     return 0;
8 | }
```

This prints:

```
10000000000000000
```

Note that our last digit has been lost!

Unsafe conversions should be avoided as much as possible. However, this is not always possible. When unsafe conversions are used, it is most often when:

- We can constrain the values to be converted to just those that can be converted to equal values. For example, an `int` can be safely converted to an `unsigned int` when we can guarantee that the `int` is non-negative.
- We don't mind that some data is lost because it is not relevant. For example, converting an `int` to a `bool` results in the loss of data, but we're typically okay with this because we're just checking if the `int` has value `0` or not.

## More on numeric conversions

The specific rules for numeric conversions are complicated and numerous, so here are the most important things to remember.

- In *all* cases, converting a value into a type whose range doesn't support that value will lead to results that are probably unexpected. For example:

```

1 | int main()
2 | {
3 |     int i{ 30000 };
4 |     char c = i; // chars have range -128 to 127
5 |
6 |     std::cout << static_cast<int>(c) << '\n';
7 |
8 |     return 0;
9 | }

```

In this example, we've assigned a large integer to a variable with type `char` (that has range -128 to 127). This causes the char to overflow, and produces an unexpected result:

```

48

```

- Remember that overflow is well-defined for unsigned values and produces undefined behavior for signed values.
- Converting from a larger integral or floating point type to a smaller type from the same family will generally work so long as the value fits in the range of the smaller type. For example:

```

1 | int i{ 2 };
2 | short s = i; // convert from int to short
3 | std::cout << s << '\n';
4 |
5 | double d{ 0.1234 };
6 | float f = d;
7 | std::cout << f << '\n';

```

This produces the expected result:

```

2
0.1234

```

- In the case of floating point values, some rounding may occur due to a loss of precision in the smaller type. For example:

```

1 | float f = 0.123456789; // double value 0.123456789 has 9 significant digits, but float
2 | can only support about 7
  | std::cout << std::setprecision(9) << f << '\n'; // std::setprecision defined in
  | iomanip header

```

In this case, we see a loss of precision because the `float` can't hold as much precision as a `double`:

```

0.123456791

```

- Converting from an integer to a floating point number generally works as long as the value fits within the range of the floating point type. For example:

```

1 | int i{ 10 };
2 | float f = i;
3 | std::cout << f << '\n';

```

This produces the expected result:

10

- Converting from a floating point to an integer works as long as the value fits within the range of the integer, but any fractional values are lost. For example:

```
1 | int i = 3.5;  
2 | std::cout << i << '\n';
```

In this example, the fractional value (.5) is lost, leaving the following result:

3

While the numeric conversion rules might seem scary, in reality the compiler will generally warn you if you try to do something dangerous (excluding some signed/unsigned conversions).



### [Next lesson](#)

**10.4** [Narrowing conversions, list initialization, and constexpr initializers](#)

4



### [Back to table of contents](#)

5



### [Previous lesson](#)

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

2

6



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!?

## 90 COMMENTS

Newest ▼



YFN

🕒 May 28, 2025 10:37 am PDT

Would converting an `unsigned double` to an `unsigned int` when you want to round it down be an example of an unsafe conversion that is done intentionally?

👍 0

↩ Reply



David

↩ Reply to YFN<sup>9</sup> 🕒 June 19, 2025 1:35 pm PDT

Wouldn't you just use `std::setprecision` if you wanna do that?

👍 0

↩ Reply



RSH

🕒 May 5, 2025 12:20 am PDT

Im boreeeddd of this chapter

👍 7

↩ Reply



Skelemen

🕒 January 31, 2025 1:50 pm PST

Hi Alex, in this chapter you said "overflow is well-defined for unsigned values and produces undefined behavior for signed values."

For what I undestand, overflow with signed values should work in the same way as unsigned valuse but they wrap around into the negatives instead.

So the question is: why is unsigned overflow well-defined but signed overflow unefined behaviour?

👍 0

↩ Reply



Alex

Author

↩ Reply to Skelemen<sup>10</sup> 🕒 February 2, 2025 9:31 pm PST

> For what I undestand, overflow with signed values should work in the same way as unsigned valuse but they wrap around into the negatives instead.

Incorrect. Signed overflow due to conversion is well defined in C++20, but signed overflow due to computation is still undefined. Many systems will just overflow into the negatives because it's the most natural behavior but that is not well-defined behavior.



My understanding is that signed overflow due to computation is undefined because there are some architectures generate an exception when signed overflow occurs due to computation.  
Also see <http://kristerw.blogspot.com/2016/02/how-undefined-signed-overflow-enables.html>

👍 1    ➡ Reply



**Sona Faris**

🕒 July 31, 2024 11:26 am PDT

“unsafe” (even though “potentially unsafe” is more accurate

I don't think it's more accurate. Potentially unsafe does mean unsafe. When we say a car is unsafe, we don't mean it will definitely crash, we mean it might crash.

👍 0    ➡ Reply



**Alex**

Author

↻ Reply to [Sona Faris](#) <sup>11</sup> 🕒 July 31, 2024 6:25 pm PDT

Fair enough. Removed the word "potentially".

👍 6    ➡ Reply



**Dongbin**

🕒 February 10, 2024 5:11 pm PST

If `3` and `3.0` are stored differently in the computer, is *value preserving* as defined in the lesson conceptual for humans (as opposed to something that can be implemented on computers)?

👍 0    ➡ Reply



**Alex**

Author

↻ Reply to [Dongbin](#) <sup>12</sup> 🕒 February 11, 2024 11:21 pm PST

Value preserving means no value data is lost in the conversion -- it isn't a statement about how data is stored/implemented.

Since computers make use of values too, I'd say it's for both. :)

👍 2    ➡ Reply



**Erad**

🕒 September 28, 2023 5:51 pm PDT

Under the five types of numeric conversions section, there are two lines of code that confound me somewhat. Here they are:

```
long l = 3; // convert int to long
```

```
long double ld = 3.0; // convert double to long double
```

Why are these essentially not 'numeric promotions' seeing that the LHS types are wider than the (types of the) RHS values?

 Last edited 1 year ago by Erad

 1  Reply



**Alex** Author

 Reply to [Erad](#)<sup>13</sup>  September 29, 2023 1:00 pm PDT

The goal of numeric promotions is to convert smaller types to larger types (generally int or double) that can be processed efficiently. Widening conversions that do not assist in this goal are not categorized as promotions.

 0  Reply



**zls**

 Reply to [Alex](#)<sup>14</sup>  October 30, 2023 8:27 am PDT

Since I have a 64 bit pc, does that mean conversion of short to double will be considered as numeric promotion? since double is 64 bits on my pc

 0  Reply



**Alex** Author

 Reply to [zls](#)<sup>15</sup>  October 30, 2023 1:03 pm PDT

No. Double is always 64-bits, even on 32-bit PCs.

 0  Reply



**zls**

 Reply to [Alex](#)<sup>16</sup>  November 3, 2023 9:21 pm PDT

- 1: A numeric promotion is the type conversion of certain narrower numeric types (such as a char) to certain wider numeric types (typically int or double) that can be **processed efficiently**
- 2: 32 bit computers are faster at processing 32 bit data types.
- 3: 64 bit computers are faster at processing 64 bit data types.
- 4: so why is short to double not numeric promotion since double would be faster to process on a 64 bit cpu (as double are 64 bits) whereas int are 32 bits
- 5: so it makes more sense for short to be converted to double as numeric promotion to me?

is there something I am missing or some misconceptions?

 0  Reply



**Alex** Author

 Reply to [zls](#)<sup>17</sup>  November 5, 2023 11:43 am PST

Conversions between type families (integral to floating point or vice-versa) are always conversions, not promotions.

👍 1

➡ Reply



**Erad**

🗨 Reply to [Alex](#)<sup>14</sup> ⌚ September 30, 2023 2:59 am PDT

So, just to be clear: although there is a widening of data type across the assignment interface from the RHS to the LHS (technically, a numeric promotion), it is NOT semantically deemed so because the LHS type is not an int or double (types that facilitate efficient processing.) Would that be an accurate explanation?

👍 1

➡ Reply



**Alex**

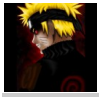
Author

🗨 Reply to [Erad](#)<sup>18</sup> ⌚ October 2, 2023 9:45 am PDT

Yah.

👍 1

➡ Reply



**Maximum G**

⌚ September 8, 2023 10:04 pm PDT

>C++20 now requires that signed integers use 2s complement. As a result, the conversion rules were changed so that the above is now well-defined as a reinterpretive conversion.

>Remember that overflow is well-defined for unsigned values and produces undefined behavior for signed values.

Aren't these 2 statements conflicting with each other?

📝 Last edited 1 year ago by Maximum G

👍 0

➡ Reply



**Alex**

Author

🗨 Reply to [Maximum G](#)<sup>19</sup> ⌚ September 10, 2023 5:50 pm PDT

No. The sign conversion rule takes precedence in sign conversion cases. In other cases, undefined behavior results.

👍 0

➡ Reply



**Ali**

⌚ August 11, 2023 12:08 pm PDT

```
1 | int u = static_cast<int>(static_cast<unsigned int>(-5)); // convert '-5' to unsigned  
   | and back
```

does this code result in any UB ?

👍 0    ➡ Reply

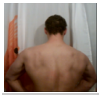


**Alex** Author

↻ Reply to [Ali](#)<sup>20</sup>    ⌚ August 12, 2023 5:17 pm PDT

In C++20, it is well-defined. Prior to C++20, it's technically UB (but I'm not aware of any modern systems on which it actually produces unexpected results). I added an advanced box discussing this.

👍 1    ➡ Reply



**Timo**

⌚ July 8, 2023 2:17 am PDT

If I understand numeric promotions correctly it means that the compiler convert data to the width of the CPU register. Any other conversion is called numeric conversion. But why isn't a float to a double called "numeric promotion" then? These days we work with 64 bit computers no?

✎ Last edited 2 years ago by Timo

👍 0    ➡ Reply

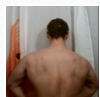


**Alex** Author

↻ Reply to [Timo](#)<sup>21</sup>    ⌚ July 9, 2023 6:15 pm PDT

Float to double is a numeric promotion.

👍 0    ➡ Reply



**Timo**

↻ Reply to [Alex](#)<sup>22</sup>    ⌚ July 10, 2023 3:44 am PDT

I seem to have skip that part when I was reading the chapter. Sorry Alex!!

👍 0    ➡ Reply



**Erad**

⌚ June 30, 2023 5:21 pm PDT

Thx Alex for your wonderful tutorial! It makes otherwise hard C++ concepts easier to grasp. I have a few comments/questions.

1. The quote below follows the Warning block in the Reinterpretive conversions section:

"Values converted using a reinterpretive conversion can be converted back to the source type, resulting in a value equivalent to the original value (even if the initial conversion produced a value out of range of the destination type)."

Referring to the last two words of this quote, do you mean 'source' instead of 'destination'?

2. In the Lossy conversions section, I think you are missing a pair of `static_cast` expressions each on line 5 and line 8.
3. In the Warning block of the Lossy conversions section, you have this quote:

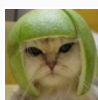
"However, there are architectures where both `int` and `double` are 8 bytes. On such architectures, `int` to `double` is a lossy conversion!"

How can it be a lossy conversion if the `int` does not have any fractional part to be converted in the first place? Or is it because, in such architectures, the `double` would have to split its available 8 bytes between the significand and the exponent? Can you please explain or show an example how `int` to `double` is a lossy conversion in this case?


4. Though not in this lesson, I read it somewhere else in this tutorial that C++ is a strongly type language. Please what does this mean?

 Last edited 2 years ago by Erad

 0  Reply



**Alex** Author

 Reply to [Erad](#) <sup>23</sup>  July 3, 2023 2:59 pm PDT

1. Yes, I meant source. Fixed!
2. Added `static_cast` to make the conversions explicit rather than implicit.
3. Yes, an 8-byte `int` uses all 64-bits to hold the integral value. An 8-byte `double` splits its bits between mantissa and exponent, so it has greater range while not being able to precisely represent all values within that range. I added an example showing data loss from this conversion.
4. There is no formal definition of "strongly typed", but strongly typed languages generally are more rigid about requiring types to be explicitly defined and will not do implicit conversions as readily. Strongly typed languages tend to have more type checking done at compile time to help ensure type safety.

 0  Reply

## Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/>
3. [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/)
4. <https://www.learncpp.com/cpp-tutorial/narrowing-conversions-list-initialization-and-constexpr-initializers/>
5. <https://www.learncpp.com/>
6. <https://www.learncpp.com/numeric-conversions/>

7. <https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/>
8. <https://gravatar.com/>
9. <https://www.learncpp.com/cpp-tutorial/numeric-conversions/#comment-610527>
10. <https://www.learncpp.com/cpp-tutorial/numeric-conversions/#comment-607213>
11. <https://www.learncpp.com/cpp-tutorial/numeric-conversions/#comment-600344>
12. <https://www.learncpp.com/cpp-tutorial/numeric-conversions/#comment-593505>
13. <https://www.learncpp.com/cpp-tutorial/numeric-conversions/#comment-587929>
14. <https://www.learncpp.com/cpp-tutorial/numeric-conversions/#comment-587966>
15. <https://www.learncpp.com/cpp-tutorial/numeric-conversions/#comment-589219>
16. <https://www.learncpp.com/cpp-tutorial/numeric-conversions/#comment-589242>
17. <https://www.learncpp.com/cpp-tutorial/numeric-conversions/#comment-589387>
18. <https://www.learncpp.com/cpp-tutorial/numeric-conversions/#comment-587989>
19. <https://www.learncpp.com/cpp-tutorial/numeric-conversions/#comment-586807>
20. <https://www.learncpp.com/cpp-tutorial/numeric-conversions/#comment-585404>
21. <https://www.learncpp.com/cpp-tutorial/numeric-conversions/#comment-583426>
22. <https://www.learncpp.com/cpp-tutorial/numeric-conversions/#comment-583487>
23. <https://www.learncpp.com/cpp-tutorial/numeric-conversions/#comment-582921>
24. <https://g.ezoic.net/privacy/learncpp.com>